# Pascal–2 V2.1/RT-11 Programmer's Guide

## Introduction

The Programmer's Guide contains nitty-gritty information about Pascal–2 for programmers well-versed in the Pascal language. This guide describes compiler commands, compilation and embedded switches, I/O control switches, and Pascal–2's low-level interaction with the PDP-11. This guide also describes ways to handle common Pascal-related implementation questions on RT-11 and contains other miscellaneous information.

This guide is **not**:

- an introduction to Pascal (see *Programming in Pascal* by Peter Grogono);
- a beginner's guide to Pascal–2 (see the User Guide);
- a detailed description of Pascal–2 (see the Pascal–2 Language Specification).

## Compiler Commands

All Pascal–2 compilation commands are divided into three parts: the compiler invocation command, the file specifications, and the compilation switches.

The compilation syntax for Pascal–2 is this:

```
.R PASCAL
*output-file,listing-file=input-files/switches
```

The R PASCAL invocation (or some other name that your system manager has chosen for the invocation command) must always come first. The '*' prompt appears for the file specifications and compilation switches.

*input-files*:
: The only required file specification is at least one input file. Multiple input files are concatenated in order, from left to right, so that a large program can be split into separate files or so that a common set of definitions can be placed in a configuration file. With "source concatenation" no input file can contain a **program** statement, except for the first file listed. If no output specification is given, the output is determined by the compilation switches; the file name is taken from the last input file specified; and the output files will be placed in the default directory. The default input file extension is .PAS. Multiple input files are separated by a comma.

*output-file*:
: The output file specifies the name of the object output, with a default extension of .OBJ. If the **macro** compilation switch is specified, the output file contains MACRO-11 code and the default extension is .MAC.

*listing-file*:
: The listing file specifies the file to receive the compilation or error listing. The default listing file extension is .LST.

If an equal sign appears on the command line, but no file name is listed in the position of the output file, no output file is generated. If no file name is listed in the

position of the listing file, a listing output is produced only if errors exist; if errors exist, output is sent to the user's terminal with the **errors** switch assumed.

*switches*: Program compilation is affected in some way by one or more of the options described in the next section. Examples in this manual show the compilation switches after the last file specification, but switches may appear after any file specification and wherever they're placed, they apply to the entire compilation. Multiple switches are separated by slashes.

## Compilation Switches

Compilation switches provide control over the files generated and over some aspects of the generated code. A switch is signified by a descriptive name (e.g., **check**). A switch name beginning with **no** reverses the effect of the switch (e.g., **nocheck**). A switch name may be abbreviated as long as the shortened form is sufficient to identify the switch. Three characters of the switch name (excluding the **no**) always identifies a Pascal-2 compilation switch (e.g., **che**, **noche**; **mac**, **nomac**).

Some switches, such as **object** and **macro**, are incompatible, causing the error message "conflicting switches specified" if used in the same compilation.

Pascal-2 compilation switches are:

### Program Options

**double** All **real** variables are in 8-byte floating-point format. You also must use colon notation (e.g., `E:18:15`) within the program to obtain double-precision values in the **write** statement. Default is "off": **real** variables are in 4-byte format. See "Extended Precision" for more details.

**pascal1** Specifies that the interface to external procedures be compatible with Pascal-1. This interface is a bit less efficient than that of Pascal-2; the **pascal1** switch should be used only when required. Default is the Pascal-2 interface.

**nomain** No main program is expected; only procedures are compiled. This switch is used most often to compile modules containing only external procedure definitions. If a main program is found, an error message is generated saying that extra statements have been found. Default is **main**: a main program is being compiled.

**own** Specifies that global-level variables are local to the compilation unit and are shared only with other external routines that have been compiled with the same program name and with **own**. Default is "off": global variables are shared.

**nowalkback**
Disables the generation of line number and procedure name tables for the procedure-by-procedure walkback that is displayed on the terminal when a program contains a run-time error. The run-time message header and error message are printed but not the walkback. Default is **walkback**: the tables are generated, and the full walkback in source terms is displayed after the message header and error message. See "Run-Time Error Reporting" later in this guide for a discussion of the walkback. The **debug** switch disables the generation of the error walkback.

### Compiler Options

**workspace:*n***
By default, the Pascal-2 compiler opens about 500 blocks of work space. The **workspace** option allows you to reduce work space to compile a small program. A realistic low range is about 150 to 200 blocks. The error message "attempt to write past eof"

on a work file indicates that the compiler needs more work space for a large program. A realistic high range is about 750 blocks.

**errors** Requests that the listing file contain only lines with errors. Unless **list** is specified, the default is "on" and errors are printed on the terminal. When **list** is specified, the default is "off" and all lines are printed in the listing file. This switch has no effect when used with the **debug** switch or the **profile** switch, because both of these switches always generate a listing file.

**list** Requests a full source listing in the listing file. If a listing file is specified, the default is "on"; otherwise the default is "off."

**debug** Requests generation of code and auxiliary files to interface with the Pascal–2 Debugger. Default is "off." The **debug** switch disables the generation of the walkback. This switch cannot be used with the **profile** switch or the **errors** switch.

**profile** Requests an execution profile when the program is run. Default is "off." The switch cannot be used with the **debug** switch or the **errors** switch.

### Code Switches

**object** Generates an object format output file with default extension .OBJ. Default is normally "on"; object code will be generated. The switch is "off" when **noobject** is specified or when no output file is provided on the command line. The switch cannot be used with the **macro** switch.

**macro** Generates MACRO-11 code in the output file. This code may be assembled by the `MACRO` assembler command to produce an object file. When **macro** is specified, **object** is set "off" and the default extension for the output file becomes .MAC. Default of **macro** is "off." The **macro** switch cannot be used with the **object** switch.

### Checking Switches

**nocheck** Disables all run-time checks, including index range checks, subrange assignment checks, pointer checks, stack checks, case label checks, and divide-by-zero checks. Note that compilation errors are still detected. Thus, if **nocheck** is specified, `var A:array [2..10] of integer; A[1] := 0;` will still be detected as a compilation error, but `I := 1; A[I] := 0;` will not be. After a program has been fully debugged, the **nocheck** switch can be used to reduce the size of the compiled code. Default is **check**.

**standard**

Requests that all Pascal–2 extended language features be flagged as errors. Default is **nostandard**.

**test** Used in debugging the compiler. Default is "off."

**times** Prints wall-clock time consumed by the compiler and the compilation rate in lines per minute. Default is "off."

### Processor Switches

The processor switch defaults to the processor option for the machine on which the compiler is running. Change the value by specifying exactly one of these four switches on the command line:

**fpp** Requests the compiler to generate code for a machine with the Floating Point Processor (FPP) option. FPP instructions include `ADDF`, `MODF`, `DIVF`, etc. This switch implies the

**eis** switch and may not be specified at the same time as the **fis** switch.

**fis** Requests the compiler to generate code for a machine with the Floating Instruction Set (FIS) option. FIS supports only the four basic floating-point instructions and is available on only a few types of machines. This switch implies the **eis** switch and may not be specified at the same time as the **fpp** switch.

**eis** Requests the compiler to generate code for a machine with the Extended Instruction Set (EIS) option. The EIS processor option includes instructions to perform integer multiplication and division. Floating-point operations will be done with calls to a floating-point simulator.

**sim** Requests code with calls to software routines for integer multiply and divide as well as for floating-point arithmetic. Should be used only if the target machine does not have EIS.

### Embedded Switches

Some characteristics of the compiled code may be controlled by switches included in the source code. These switches take the form of a Pascal comment beginning with a dollar sign '$' and followed by a descriptive name, for example:

```
{$indexcheck}
```

A switch name beginning with "no" reverses the effect of the switch, for example:

```
{$noindexcheck}
```

Most switches may be abbreviated to a minimum of three characters, for example:

**{$ind}** or **{$noi}**

However, when using **$nopointercheck** and **$noprofile** be sure to enter more than three characters, or the compiler will treat the switch as an ordinary comment.

Multiple switches can be embedded within a single comment. The switches must be separated by commas; only the first may have the dollar sign. The following forms are equivalent:

```
{$noindex,norange}
{$noindex}{$norange}
```

Embedded switches are counting switches. Each occurrence increments or decrements the switch value; the switch is enabled if its value is greater than zero. The initial value of a switch is controlled by an equivalent compilation switch, such as **debug**, if the equivalent compilation switch exists. If no equivalent switch is present on the command line, the initial value is determined by the defaults described below.

Once set, some switches are valid for the entire program, as with **$own**. In some cases, the "**no**" form of the switch is the one normally used, as with **$nomain**.

Some switches may be turned "on" and "off" for a particular section of code, either on a statement-by-statement or procedure-by-procedure basis. The following example shows how debugging can be

turned off for a procedure:

```
  :
{$nodebug} ——————————————————————— debugging turned off

procedure P;
  begin
  : ———————————————————————————— body of procedure P
  end;

{$debug} ——————————————————————— debugging enabled again
  :
```

The particulars of each switch are described in the following sections.

**Program Options**

**$double** Specifies that all real arithmetic is to be done with double precision rather than with single precision. **$double** applies to the entire compilation. You also must use colon notation (e.g., `E:18:15`) to print the double-precision values in a **write** statement. This switch must appear in the program before any data of type **real** is defined or used. Default is "off."

**$pascal1**

Specifies that external procedures will be called in a manner compatible with Pascal–1. This switch may slow program execution but should simplify conversion of programs from Pascal–1 to Pascal–2. The default is "off."

External Pascal–2 procedures may be called regardless of the setting of this switch.

**$nomain** No main program is expected; only procedures are compiled. This switch is used most often to compile modules containing only external procedure definitions. If a main program is found, an error message is generated saying that extra statements have been found. Default is **$main**: a main program is being compiled.

**$own** Specifies that global-level variables are local to the compilation unit and are shared only with other external routines that have been compiled with the same program name and with **$own**. **$Own** applies to the entire compilation. Default is "off": global variables are shared.

**$nowalkback**

Disables the generation of line number and procedure name tables for the procedure-by-procedure walkback that is displayed on the terminal when a program contains a run-time error. The run-time message header and error message are printed but not the walkback. Default is **walkback**: the tables will be generated, and the full walkback in source terms will be displayed following the message header and error message. The **debug** compilation switch disables the generation of the error walkback. See "Run-Time Error Reporting" later in this guide for a discussion of the walkback.

**Compiler Options**

**$nodebug, $debug**

Disables/enables some of the overhead of the Pascal–2 Debugger. These two switches will have effect only when the **debug** compilation switch is specified. The **debug** switch generates the extra files needed for debugging and sets the **$debug** switch "on."

**$Nodebug** can be used to turn off some of the debugging overhead for procedures or functions that have already been fully tested. **$Debug** can be used to restore debugging for other procedures.

**$noprofile, $profile**
Disables/enables some of the overhead of the Pascal–2 Profiler. These two switches will have effect only when the **profile** compilation switch is specified. The **profile** switch generates the extra files needed for profiling and sets **$profile** "on." **$Noprofile** can be used to turn off profiling for procedures or functions that do not need to be profiled, and **$profile** can be used to restore profiling for other procedures.

When the **begin** statement of a procedure is compiled, the state of the **$debug/$nodebug** and **$profile/$noprofile** switches will determine debugging or profiling for that entire procedure. Note that a procedure constitutes the smallest section of code that can be debugged or profiled; you can't debug or profile individual lines of a procedure.

The **$debug/$nodebug** and **$profile/$noprofile** switches serve the same functions as far as the code generated. You would never use both sets in the same compilation. (You can't debug the program and profile it at the same time.)

**$nolist** Turns off the listing of source lines in the listing file; **$list** restores the listing of source lines. The switch may be turned on or off after each line of source code. The listing file will display the **$nolist/$list** switches, and the line numbers will reflect the lines for which listing has been disabled. In this program fragment, listing has been disabled on lines 3 through 5:

```
1       program Ex(output);
2       {$nolist}
6       {$list}
7
8           begin
              .
              .
              .
```

Lines with errors will be displayed even if the **$nolist** switch is on. Default is **$list**.

Do not use the **$nolist** switch during debugging sessions. If you attempt to access any "unlisted" line(s), the response will be the message "No such statement in this procedure." Other errors also may be produced.

**$standard**
Like the corresponding compilation switch, **$standard** causes all extended language features of Pascal–2 to be flagged as compilation errors. By using the embedded switch at the beginning of the program, you don't have to use the **standard** switch every time you compile the program.

In addition, if you want to compile the program using language extensions of Pascal–2, but you want to mark the non-standard features (for later transportability to another compiler, perhaps), insert the **$standard** switch at the start of the program, and enclose any non-standard sections with the switches **$nostandard** and **$standard**. The compiler will then check the rest of the program for non-standard features, so that you can minimize your use of extensions. The **$nostandard** switch will be a textual flag to aid any future conversion to a standard program.

The **$standard** and **$nostandard** switches may be turned on or off after each line of source code. Default is **$nostandard**, which accepts the extended language features of Pascal–2 as correct forms.

### Run-Time Checking Switches

The compilation switch **nocheck** will turn off all run-time checks. The embedded switches listed below will cancel the particular checks listed below. Any of these switches can be placed at the start of the program to turn off a particular kind of check throughout. Or, "on/off" pairings can be used on a statement-by-statement basis within the program.

Turning off run-time checks will reduce the size of the program. However, we recommend that you do not turn off any checks until the program has been fully debugged.

**$noindexcheck**
: Stops generation of code for array bounds checks; no array index is checked as to whether it is within the array bounds. Default is **$indexcheck**.

**$nopointercheck**
: Stops generation of code that checks for **nil** or invalid pointer values. Default is **$pointercheck**.

**$norangecheck**
: Cancels the subrange assignment and **case** statement check capabilities. No assignment to a variable of subrange type is checked as to whether the assigned value is within the allowed range. Also, **case** selectors are not checked for matching labels. Default is **$rangecheck**.

**$nostackcheck**
: Stops the generation of code for stack overflow checks on procedure and function entry. No entry to a procedure or function is checked as to whether adequate stack space is available for local variables. Note that some procedures call support library routines that check for stack overflow. Thus, even when compiled with this switch, some programs may still report "stack overflow" errors. The default is **$stackcheck**.

### Compilation Examples

The following examples show the effects of various switches on the compilation.

Example 1.

```
.R PASCAL
*PROG/LIST
```

Compiles the file PROG.PAS and generates an object file PROG.OBJ and a listing file PROG.LST. The **check** switch is assumed to be on, and code will be generated for the hardware options of the machine on which the program is being compiled.

Example 2.

```
.R PASCAL
*PROG,PROG=PROG
```

Equivalent to Example 1.

Example 3.

```
.R PASCAL
*PROG=PROG/NOCHECK/FIS
```

Compiles the file PROG.PAS and generates an object file PROG.OBJ. Any errors will be listed on the user's terminal. No run-time checking code is generated, and code will be generated for a CPU with FIS instructions.

Example 4.

```
.R PASCAL
*HEADER,MIDDLE,PROCED/NOMAIN
```

Concatenates and compiles the files HEADER.PAS, MIDDLE.PAS, and PROCED.PAS in the order given, and generates an object file, PROCED.OBJ. This code has no main body and therefore contains external procedures. The **check** switch is assumed to be on, and code will be generated for the hardware options of the machine on which the program is being compiled.

Example 5.

```
.R PASCAL
*_,TT:=PROG
```

Produces a listing file to the terminal but no PROG.OBJ file.

### Linking and Executing

After compilation, Pascal-2 programs must be linked to the support library before being executed. The DCL sequence is:

```
.LINK PROG,SY:PASCAL
.RUN PROG
```

The preceding two commands may also be entered in following manner as well:

```
.R LINK
*PROG = PROG,SY:PASCAL
*^C
.RUN PROG
```

See "The Linker, Overlays, and the Librarian" for details of the link process.

## I/O Control Switches

The **reset** and **rewrite** standard procedures accept two additional arguments specifying the file name of an external file and default fields of the file name. These arguments can also include I/O control switches that give explicit control of the operating system interface details. (A fourth parameter can also be specified. See the Language Specification for a complete discussion of **reset** and **rewrite**.)

The I/O switches appear in the file name or default name parameters as shown in these examples:

```
rewrite(F,'data.dat/seek/span/size:12.',ErrStatus);
```

A special device (TI:) also may appear in the **reset** and **rewrite** calls. The TI: device connects to the Pascal–2 terminal driver and is used in place of the TT: driver for interactive use.

A complete list of I/O switches appears below, followed by individual details. All switches may be abbreviated to the first two letters. The parameter *n* is a octal number unless terminated by a period, in which case it is a decimal number.

| Switch | Meaning |
|---|---|
| **/buffersize**:*n* | Allocate *n* bytes for buffer (hereafter abbreviated **/buff**) |
| **/go** | Allow programmed error handling |
| **/nfs** | Non-File-Structured access |
| **/odt** | Single-character terminal input |
| **/seek** | Direct-access file |
| **/size**:*n* | File storage allocation |
| **/span** | Records span block boundaries |
| **/temp** | Temporary file |

**/buff**:*n* (Buffersize): The **/buff**:*n* switch specifies the storage to be allocated to a file buffer. Pascal–2 normally allocates the minimum space required for a file buffer, 512 bytes. For disk files this default value may be raised by multiples of 512 to improve the efficiency of I/O transfers, at the cost of additional memory. Line-oriented files such as terminal and line-printer files receive buffer space equal to the width of the line, usually 80 characters. Efficient single-character I/O requires a minimum of buffer space, 1 or 2 characters. See also the **/odt** switch below.

The value of *n* used with **/buff**:*n* is a decimal number if terminated with a period; otherwise octal.

**/go** (I/O Error Continue): I/O transfer errors are normally fatal and cause immediate program termination. The **/go** switch indicates that I/O errors on the specified file are non-fatal and allow the program to continue executing. This switch is the equivalent of **noioerror**, one of three predefined Pascal procedures that trap and respond to I/O errors. See the section "I/O Error Trapping." The switch **/go** has been left in for compatibility with previous implementations.

**/nfs** (Non-File-Structured Access): The **/nfs** switch is used to achieve non-file-structured access to a device that is normally file-structured, such as a disk device. Because direct access to such a device can destroy its directory structure, Pascal–2 prevents non-file-structured access unless the **/nfs** switch is used.

**/odt** (Single-Character I/O): The **/odt** switch derives its name from the ODT Debugger (Octal Debugging Technique), which is driven by single-character commands. The **/odt** switch is used with keyboard files, indicating that each character read from the file is to be processed immediately without any wait for a carriage return or other action

character. The **/odt** switch is only effective for files on the `TI:` terminal device, and the size of the buffer should be reduced to a minimum, as shown:

```
reset(input,'ti:/odt/buff:2');
```

Note that the `RUBOUT` and Control-U (^U) keyboard editing capabilities are not effective with **/odt**.

**/seek** (Direct-Access): The **/seek** switch enables the use of the direct-access **seek** procedure, and it permits both read and write access to the file variable so that records may be updated with calls to **get** and **put**. The **seek** procedure is described in the Language Specification.

**/size:***n* (File Allocation): The **/size:***n* switch used in the **rewrite** procedure specifies the space to be allocated for the file. The size of the file is given in blocks of 512 bytes.

**/span** (Spanned): In files created or accessed by Pascal–2 programs, fixed-length records are normally "blocked." This means that an integral number of records are stored in one disk block of 512 bytes, with any remaining storage in that block being unused. The **/span** switch packs records more efficiently, with records spanning from one disk block to the next. This requires additional buffer memory, which is automatically allocated, and some additional computation. Spanned and blocked files are not generally compatible. Files created with **/span** should be read with the same switch.

**/temp** (Temporary): This switch is used with **rewrite** to indicate a temporary file that will be deleted on termination. No file name is needed if this switch appears.

## External Modules

Pascal–2 implements separate compilation through the concept of an external module, a program fragment containing at least one procedure or function. External modules are compiled independently of other program units and combined by the Linker. External modules may be stored in libraries to simplify the handling of common routines.

External modules may reference global variables shared by all of the modules making up a program. If each module (including the main program) is compiled with the same global variables, the effect is as if all modules were compiled together. For this to work properly, the global declarations in the external procedure and main program must contain the same variable declarations in the same order. Parameter lists also must agree (i.e., contain the same parameter declarations in the same order). The simplest and most efficient way to do this is to place all global declarations, including references to external procedures and functions, in a header file then include the header file in the external module and in the main program, using the **%include** compiler directive (see "Multiple Source Files").

External modules must be referenced at the outermost (global) level of a main program, but they may be called from any point in the code. An external module compilation requires either the **nomain** compilation switch or the **$nomain** embedded switch. Both switches specify that no main program is contained in the source file. The **nomain** switch is specified on the command line, whereas the **$nomain** embedded switch is placed at the beginning of the external module.

An external procedure name consists of the first six characters of the procedure or function identifer. External procedure names must uniquely identify an external routine because they are used as global symbol names by the Linker.

Linking errors most pertinent to external modules are:

- Duplication of external symbols. An external procedure has been defined in more than one module. When multiple definitions are encountered, the Linker uses the first definition only and ignores succeeding definitions.
- Undefined external symbols. The program has referenced an external routine that was not defined in any of the object files or libraries specified on the command line. This error indicates that the program contains unresolved global references.

In each case, the Linker responds with an error message and produces an output file that cannot execute (permission bits on the file are not set).

Two compiler directives, **external** and **nonpascal**, allow the use of external modules. The **external** directive defines a procedure or function implemented in Pascal–2 as "external," which means that the procedure may be referenced by other modules and that both the external module and the program or module that calls it expects to find the normal Pascal–2 calling sequence of parameters on the stack. The **nonpascal** directive defines a routine written in a language other than Pascal, such as FORTRAN or MACRO-11, and generates a call to the FORTRAN interface routine (P$111) in the Pascal support library. P$111 creates the standard DEC calling sequence of parameters which is expected by the external module, and which differs from Pascal–2's.

### CAUTION

Observe two cautions when using the **external** or **nonpascal** directive. Parameters to external routines cannot be checked by the compiler for type conformance across module boundaries, so an accidental type mismatch may cause unpredictable results. Also, the compiler cannot verify the conformance of global data. As mentioned above, use of the **%include** directive can help reduce problems in this area.

### Calls to Pascal-2 Routines

The syntax of the **external** directive is similar to the syntax of the **forward** directive in that it consists of two distinct parts: the declaration and the body. The declaration includes the external procedure or function name and the argument list, followed by the **external** directive.

```
procedure GetString(Arg: Argtype);  ------------- this is the declaration
  external;  ------------------------------------ external directive is required
```

This declaration must appear in the external module and in each compilation unit that calls that external routine. You can place the declaration in a header file and use the **%include** directive to insert it appropriately.

The body of the external module contains the actual code for the procedure or function and must not include an argument list.

```
procedure GetString;  --------------------- this is the procedure body
  begin
    :
  end;
```

The body and declaration must be compiled together in order for the external procedure to function properly. The external procedure may then be called in the same way as any other procedure or function:

```
GetString(Length);  ------------------------ external procedure call
```

### Example Using External Directive

The practical application of external procedures is best shown by example. The following sample illustrates the declaration and use of external procedures and the correct way to access global variables. Note that the global declarations must be identical in the external procedure (CHANGE.PAS) and in the main program (MAINLINE.PAS). Note also the use of the **$nomain** embedded switch in the external procedure.

First, we create a separate header module HDR.PAS containing the external procedure reference and the program's global declarations.

HDR.PAS

```
type  ------------------------------------------ global declarations
  GlobalType = record
      B: boolean;
      V: integer;
    end;
var
  Glob: GlobalType;
  I: integer;

procedure Change(P: integer);
  external;  ---------------------------------- external directive must appear
```

The file CHANGE.PAS consists of the external procedure **Change** and the **%include** directive, as follows:

CHANGE.PAS

```
{$nomain}  ——————————————————————————  embedded switch
%include hdr.pas  ———————————————————  header module
procedure Change;  ——————————————————  no parameters here
  begin
    with Glob do
      begin  ———————————————————————  change global variables
        B := true;
        V := V + P;
      end;
  end;
```

The external procedure is then compiled with the **nomain** option embedded, using the command:

```
.R PASCAL
*CHANGE
```

However, you can omit the embedded option and specify the **nomain** on the command line in the following manner:

```
.R PASCAL
*CHANGE/NOMAIN
```

The externally defined procedure **Change** may now be called within any program unit that includes HDR.PAS and is linked with CHANGE.OBJ. For example, assume that the file MAINLINE.PAS consists of this:

MAINLINE.PAS

```
program Mainline;
  %include hdr;  ———————————  external procedure and global declarations

  procedure Before;

    begin
      with Glob do
        writeln('Before executing Change, B = ',B,' and V =',V:2,'.');
    end;

  procedure After;

    begin
      with Glob do
        writeln('After executing Change, B =',B,' and V = ',V:2,'.');
    end;

  begin      { program Mainline }
    with Glob do
      begin  ———————————————————————  initialize global variables
        B := false;
        V := 0;
      end;
    I := 45;
    Before;
    Change(I);  ————————————————————  the external call
    After;
  end.
```

Compile MAIN as any other main program and link the main program and external module, as shown:

```
.R PASCAL
*MAIN
.R LINK
*MAIN,MAIN=MAIN,CHANGE,SY:PASCAL
```

Running MAIN yields these results

```
.RUN MAIN

Before executing Change, B = false and V = 0.
After executing Change, B = true and V = 45.
```

### Calls to Non-Pascal-2 Routines

The **nonpascal** directive is used instead of **external** when the external procedure is generated by an assembler or a compiler other than Pascal-2. **Nonpascal** creates an interface between the Pascal-2 calling sequence generated by the main program or module and the standard DEC calling sequence required by FORTRAN and most MACRO routines. In addition, when the **nonpascal** directive is invoked, register **R5** is used as a pointer to the list of parameters.

Syntax for the **nonpascal** directive consists of a separate declaration and body. The declaration contains the name of the procedure or function and the argument list, followed by the **nonpascal** directive.

#### Calling MACRO Subroutines

The external declaration for a MACRO function looks like this:

```
program Test;

var i: integer;

function afunct (var i: integer) : integer;     ———————— declaration of
  nonpascal;                                              MACRO routine

begin  ——————————————————————————————————————— body of the main program
  i := 10;
  writeln(afunct(i));
end.
```

The MACRO routine AFUNCT is written:

```
    ; Returns argument plus 10


AFUNCT::
        mov     @2(r5),r0
        add     #12,r0
        rts     pc
        .end
```

Type matching for the declaration and use of parameters are the user's responsibility.

The sample program Test is compiled with the command:

```
.R MACRO
*AFUNCT
.R PASCAL
*TEST
.R LINK
*TEST,TEST=TEST,AFUNCT,SY:PASCAL
.RUN TEST
     20
```

MACRO routines written with the Pascal-2 PASMAC utility must be declared as **external** rather than **nonpascal**, because PASMAC simulates the Pascal-2 calling sequence.

**Calling FORTRAN Subroutines**

For Pascal programs to call FORTRAN subroutines, two conditions must be met: all parameters must be passed by reference, not by value, and FORTRAN subroutines must be declared in Pascal programs with the **nonpascal** directive.

The program FTEST.PAS shows a way to call FORTRAN subroutines from a Pascal program. The program reads three integers from the terminal, calls the FORTRAN subroutine ADDEM to calculate the sum of the three numbers, then prints the sum.

Subroutine ADDEM.FOR contains these statements:

```
      SUBROUTINE ADDEM(A,B,C,D)
C
C  ADDS A, B, C TO PRODUCE D.
C
         IMPLICIT INTEGER (A-Z)

         D = A + B + C

         RETURN

      END
```

The main Pascal program contains these lines:

```
program FORtest;
  var
    A, B, C, D: integer;

    procedure ADDEM(var A, B, C, D: integer);
      nonpascal;

  begin
    write('Enter 3 values: ');
    readln(A, B, C);
    ADDEM(A, B, C, D);    { add the numbers}   ————— FORTRAN call
    writeln('The answer is ', D);
  end.
```

Compile and run the program using these commands. Assume that the FORTRAN subroutine ADDEM has already been compiled.

```
.R PASCAL
*FTEST
.R LINK
*FTEST=FTEST,ADDEM,SY:PASCAL
.RUN FTEST
Enter 3 values: 20 40 120
The answer is      180
```

Two restrictions relate specifically to the use of FORTRAN subroutines in Pascal programs. First, Pascal-called FORTRAN subroutines cannot access files opened in the Pascal program. However, these FORTRAN routines can use files that they themselves open.

Second, FORTRAN allows the passing of "null" parameters to subroutines, in which a comma is used as a place holder for an optional parameter. Pascal has no such feature. To pass null parameters to a FORTRAN subroutine from Pascal, use the **origin** directive to declare the null parameter. The variable and procedure declaration for the Pascal program appears as shown:

```
var
  ListNumber, LastList: integer;
  INull origin 0: integer;   - specifies null integer parameter for FORTRAN
  RNull origin 0: real;      —— specifies null real parameter for FORTRAN
  Rewind: boolean;

procedure FPREW(var Number, Last: integer;
                var I: integer; var R: real;
                var Rew: boolean);
nonpascal;
```

The FORTRAN subroutine declaration is then:

```
subroutine FPREW(Number,Last,I,R,Rew)
```

When you call the FORTRAN subroutine from the Pascal program, substitute the appropriate **Null** variable for any unnecesary parameters. In the case of FPREW, the third and fourth parameters are null parameters:

```
FPREW(ListNumber,LastList,INull,RNull,Rewind);
```

## External Module Libraries

Suppose you want a library of procedures that can be referenced by any program. For a particular program, you do not necessarily reference all the procedures in that library, and you do not want the entire library loaded with the program.

Procedures and functions from one compilation unit form a single object file and cannot be selectively loaded. For example, if procedures A, B, and C are compiled together and placed in a library, any reference to one of them causes all three to be loaded. On the other hand, if each procedure A, B, and C is compiled separately and the three object files are placed in the same library, then a reference to one of them causes only that procedure to be loaded in the program. To keep final program size to the minimum, library procedures should be compiled separately whenever possible.

Rather than having an external declaration in the main program for each procedure needed, create a single "header" file containing the external declarations for all the external procedures defined in the library. This header file can be included in the compilation with the `$include` directive placed near the beginning of the program source file. No external reference is generated for any external procedure in the header file that is not used by the program, so only those procedures actually used by each compilation unit are loaded into the final image file (assuming that the library procedures were themselves separately compiled). See "Multiple Source Files" for use of the `$include` directive.

By using a header file in this way, you can avoid errors that could be caused by a mismatched declaration, forcing any change made to a declaration for an external procedure to be reflected in all programs using that procedure. Carried to the fullest extent, a library and its corresponding header file can be used system-wide.

# The Linker, Overlays, and the Librarian

## The Linker

Object modules produced by the Pascal–2 compiler are compatible with object modules produced by the MACRO assembler, FORTRAN compiler, and other RT-11 system utility programs. The Linker, LINK.SAV, can produce overlaid executable programs, allowing much larger programs than the 32K-word address space. (Overlays are explained below.) The Librarian, LIBR.SAV, can build libraries of object modules. Some highlights of Linker and Librarian capabilities are covered here. See the *RT-11 System User's Guide* or the *RT-11 System Utilities Manual* for details.

To run the Linker, give the command:

```
.R LINK
*
```

('*' indicates that the Linker is waiting for a command.)

The first command line can include the output file, a map file if desired, and up to six input files. The file PASCAL.OBJ, which contains the Pascal run-time library, must be used when linking a Pascal program. The example below links the object module `MAIN` with two others, `SUB1` and `LIB1`, to produce a "privileged" MAIN.SAV file and a MAIN.MAP file. The job is privileged in that the monitor, device handlers and I/O page are mapped into the program's address space, as opposed to being excluded from the program's memory (a "virtual" job, see below).

```
*MAIN,MAIN=MAIN,SUB1,LIB1/C
*SY:PASCAL
*^C
```

An alternative to the `R LINK` command is `LINK`. `LINK`, a DCL command, allows you to enter the command line on the same line as the command, as in:

```
.LINK MAIN,SUB1,LIB1,SY:PASCAL
```

### Virtual Jobs and the XM Monitor

The XM monitor has the ability to create a "virtual job" with a separate 32K word address space. A virtual job does not need to reserve space for the monitor, device handlers, or the I/O page, and can therefore use the entire available address space. A virtual job cannot make direct reference to device registers or perform other specialized functions. These operations are restricted to "privileged jobs." For most purposes, virtual jobs are preferred.

Making a virtual job requires the setting of bit 10 (2000 octal) in the JSW (Job Status Word) in location 44 (octal) of the .SAV image file. The file VIRJOB.OBJ, supplied with Pascal–2, sets the virtual bit in the JSW for Pascal–2 or other programs. VIRJOB should be included as an input file in the Linker command(s).

```
.R LINK
*MAIN = MAIN,SUB1,LIB1,VIRJOB/C
*SY:PASCAL
*^C
```

One additional restriction: The .SAV image file containing a virtual job must be stored on the system device (`SY:`), and must be run via the `R` command.

## Overlays

Overlaying is the method of dividing a program logically into small pieces called "segments," allowing you to execute a program that is larger than the available memory (32K). Each overlaid program has a root segment that is always resident in memory and any number of executing overlay segments. Each segment is assigned to a particular area of memory called an overlay region, with each overlay region containing one or more segments.

A program may require (further) overlaying if run-time errors such as "not enough memory" or "stack overflow" cause the program to abort, or when the program cannot be loaded into memory because it is too large.

The Linker is capable of creating two kinds of overlay structures, low memory overlays on the SJ, FB and XM monitors and extended memory or "virtual" overlays on XM. With low memory overlays, only the root and the executing segment reside in the program's 32K address space; all other segments reside on disk and are called in and overlaid against the previous segment, overwriting it. Extended memory overlays require the extended-memory features of the XM monitor to create a physical address space greater than 32K words for the program. This address space is made up of segments assigned to specific virtual overlay regions. Once called into virtual memory to execute, a virtual overlay region remains in physical memory, thus reducing the number of disk accesses to load in the next segment.

In most cases, low memory and extended memory overlays may be mixed for added program flexibility. Details of both types of overlays are given below, oriented toward Pascal tasks. The full overlay capabilities are described in the *RT-11 System Utilities Manual.*

**NOTE**

> The Linker uses channel 15 as the overlay load channel; for this reason, your program should not access channel 15 directly. See the "Support Library" section for information on channels.

### Low Memory Overlays

The `/O:`*n* (Low Memory Overlay) option selects the low memory overlay facilities of the Linker, where the parameter *n* indicates the overlay region number. Sets of modules allocated to the same region will be overlaid against other modules in the same region, with only one set of modules per region actually in memory at any one time.

The following sequence links a main program and several external modules into an overlaid executable file. The main program and the Pascal library are not overlaid and must be in the root segment (on the first command line). `FIRSTA` and `FIRSTB` do not call each other and are overlaid against each other in region 1. `TWOA` and `TWOB` do not call each other and are overlaid against each other in region 2. The set of `NEXTA`, `NEXT2A`, and `NEXT3A` are overlaid against the set of `NEXTB`, `NEXT2B`, and `NEXT3B` in region 3. No module in one set calls a module in the other set that is overlaid in region 3. The continue option (`/C`) allows the input file list on the next line to be included in the linking.

```
.R LINK
*PROG = MAIN,SY:PASCAL/C
*FIRSTA/O:1/C
*FIRSTB/O:1/C
*TWOA/O:2/C
*TWOB/O:2/C
*NEXTA,NEXT2A,NEXT3A/O:3/C
*NEXTB,NEXT2B,NEXT3B/O:3
*^C
```

You can also use the command **LINK/PROMPT**, with the same effect.

**Extended Memory Overlays**

The **/V:***n***:***m* (Extended Memory Overlay) Linker option is used to create extended memory overlays where *n* is the overlay region number and *m* is the optional partition into which the segment is loaded. Both privileged and virtual jobs can use extended memory overlays. The **/V:***n***:***m* option produces a load image that no longer requires the I/O page and monitor to reside in the program's address space. This makes the entire address space available to the program, an advantage for programs with large root segments or with high demand for dynamic memory (stack and heap).

The next example uses the same program files as in the preceding overlay example to create a virtual job. Note that the **/V:***n***:***m* option replaces the **/O:***n* option used above.

In addition to the main program and support library, the root contains the VIRJOB code, making the program a virtual job. For comparison, refer to the example in the next section in which the root is a null segment and the main program and support library are loaded into the first (and only) virtual overlay region.

```
.R LINK
*PROG = MAIN,VIRJOB,SY:PASCAL/C
*FIRSTA/V:1/C
*FIRSTB/V:1/C
*TWOA/V:2/C
*TWOB/V:2/C
*NEXTA,NEXT2A,NEXT3A/V:3/C
*NEXTB,NEXT2B,NEXT3B/V:3
*^C
```

**Linking a Program Whose Root Exceeds 16K**

Due to a restriction in the XM monitor, Pascal programs with root segments or mainlines larger than 16K words cannot be loaded at all. This restriction applies to both privileged and virtual jobs. In the case when overlaying, or further overlaying, of the program is undesirable, use the file START.OBJ from the Pascal support library to create a "null" root segment. This places the program code in the first (and only) virtual overlay region. In the process excess memory in the root is added to the heap's free list, increasing the amount of dynamic memory available to the program.

START contains code necessary to create a null root. START defines two "grow" psects, **P$GROW** and **P$GRWH**, which are used in the creation of the 4K-word null root segment containing low core and vector information ($1000_8$ words) and START (32 words).

The **/U:20000** (Round) Linker option expands **P$GROW** to the root's upper boundary (**P$GRWH**). At run time, the Pascal support library places this excess region on the heap's free list for use by the Pascal program.

When START is used, any number of segments but only one virtual overlay region can be specified in the overlay structure. Programs having more than one virtual overlay region cannot be linked in this way. As the next example shows, each overlaid module is placed in its own partition within the virtual overlay region. The overlaid module is loaded into virtual memory when first called and resides there for the remainder of the execution, reducing the number of disk accesses. The **V:1:1** option creates virtual overlay region 1, partition 1, containing the tables and code for program TEST and the Pascal support library. The START and VIRJOB code is placed in the root.

Example:

```
.R LINK
*TEST = START,VIRJOB/U:20000/C
*TEST,SY:PASCAL/V:1/C
*FIRSTA/V:1:2/C
*FIRSTB/V:1:2/C
*TWOA/V:1:3/C
*TWOB/V:1:3/C
*NEXTA,NEXT2A,NEXT3A/V:1:4/C
*NEXTB,NEXT2B,NEXT3B/V:1:4/C
*//
Round section? P$GROW
*^C
```

For more information on virtual jobs and overlays, see the *RT-11 Software Support Manual* and the *RT-11 System Utilities Manual.*

### The Librarian

The Librarian combines relocatable object module files to form an object module library. This library may be included as input to the Linker, which will select only those modules needed by the program being linked. Note that a module always consists of the entire set of procedures and functions from its compilation. Individual procedures cannot be selected from a module.

For example, the dynamic string package STRING.PAS, supplied as a Pascal–2 utility, can be edited to form 12 files, with each file containing one procedure or function. The 12 files can then be compiled and combined into a library containing 12 modules, as follows.

```
.R LIBR
*STRING=LEN,CLEAR,READS,WRITES,CONC,SEARCH/C
*INSERT,ASSIGN,ASCHAR,EQUAL,DELETE,SUBS
*^C
```

## Extended Precision

Values of type **real** are normally stored in the PDP-11 single-precision format, which requires 2 words of storage per value and offers about 7 decimal digits of precision. The **double** compilation switch or the **$double** embedded switch gives double precision to all **real** values. Each extended-precision value occupies 4 words of storage and provides approximately 15-digit precision in all **real** calculations, including the transcendental functions.

Normal and extended-precision values cannot be mixed in a program: the **double** or **$double** switch generates extended precision for all **real** values. All external modules must be compiled with the same precision as the main program, even if no **real** variables are present.

In addition, you must use the colon notation output format (e.g., `E:18:15`) to display double precision values in **write** statements.

## Support Library

The Pascal support library is a collection of modules contained in an object module library called PASCAL.OBJ located on the system device. When compiling a program, the Pascal compiler generates subroutine calls to routines in the Pascal support library. The Linker places these routines in the run-time library. The entry points in the library are identified as p$*nnn* where *nnn* is a small integer. Appendix E of this guide contains a list of these support library entry points. To see these subroutine calls, inspect the MACRO-11 code generated by the Pascal-2 **macro** switch. Support library routines not called by the compiler have a name instead of a number following the p$.

Most of the routines in the Pascal support library perform I/O operations or arithmetic computations such as floating-point simulation or trigonmetric function approximation. Other routines allocate dynamic memory and report error conditions. Still other routines allow you to change the run-time error reporting to suit your needs. When you build a Pascal job, the Linker searches the Pascal support library for the modules required to run the job. For example, if you compute a logarithm in your program, the Linker includes the support library module that approximates logarithms ($FLOG, which defines the entry point p$102).

In most Pascal tasks, the Linker includes from 3K words to 9K words of library modules.

### Initializing the Support Library

The Pascal support library is initialized at the start of a Pascal program. When a typical execution begins, the system transfers control to p$bgn, the transfer address of the program, and the support library initialization procedure p$59 is called. This procedure initializes global variables used by the support library; then it requests that the operating system expand the program code by 4K bytes to make room for the stack, which is originally located in low memory. The routine then moves the stack from low memory to its new location at the end of the program code.

After the stack is repositioned, control transfers to p$33, the file initialization routine. P$33 assigns the standard files `input` and `output` (channel 16 and 17 for both) to `TT:`, the terminal. (Channels are discussed below.) The support library then transfers control to the first statement of the program, and execution begins. However, if the program is being debugged with the Debugger, instead of control transferring to the program, control transfers to p$67, the Debugger initialization routine. This routine initializes the Debugger and opens its files. The Debugger then takes over execution of the program. (See the Debugger Guide for details.)

### Support Library Data Definitions

Constants, data and internal file definitions used by the support library are contained in the file LIBDEF.PAS, included in the Pascal-2 distribution kit. In addition to its use with the support library, this file is "included" in the error-reporting module OPERRO.PAS. LIBDEF defines the file variable, the file status block and the library data area.

By using the `%include` directive, users can include LIBDEF.PAS in any program that accesses the support library's work area directly. The example program PCHAN.PAS, below, defines a function named `GetChannel`, which returns the channel number associated with an open file. The program prints the channel numbers for standard files `input` and `output` and for two other files opened by the program. Note that the support library stores the channel number times 2 because RSTS requires all

channel numbers be doubled. Also noteworthy is the function **loophole**, predefined in the compiler and used in **GetChannel**. See the Language Specification for details on **loophole**.

```
program PrintChannel;
%include 'libdef';
  const
    Firstfile = 'FILE1.DAT';
  var
    X, Y: text;
    Filename: packed array [1..10] of char;

    function GetChannel(var N: text): integer;
      var
        F: user_file_variable;      ———————— data type defined in LIBDEF.PAS

      begin   { procedure GetChannel }
        F := loophole(user_file_variable,N);
        GetChannel := F^.channel div 2;   { RSTS uses channel * 2 }
      end;   { procedure GetChannel }

  begin  { program PrintChannel }
    writeln('Input  is open on channel: ', GetChannel(input));
    writeln('Output is open on channel: ', GetChannel(output));
    rewrite(X,Firstfile);
    writeln('File 1, ', Firstfile,', is open on channel: ', GetChannel(X));
    write('Enter a file name: ');
    readln(Filename);
    reset(Y,Filename);
    writeln('File 2, ',Filename,', is open on channel: ', GetChannel(Y));
  end.   { program PrintChannel }
```

To compile and execute the program, use this command:

```
.R PASCAL
*PCHAN

.R LINK
*PCHAN=PCHAN,SY:PASCAL
*^C
.RUN PCHAN
Input  is open on channel:        16
Output is open on channel:        17
File 1, FILE1.DAT, is open on channel:        15
Enter a file name: FILE2.DAT
File 2, FILE2.DAT, is open on channel:        14
```

**NOTE**

The definitions in LIBDEF.PAS are provided for informational purposes and are subject to change with each release of Pascal-2. Users who desire a more detailed description of the internal workings of the Pascal support library must obtain the library sources.

## Support Library's Use of Channels

An I/O channel is the means by which RT-11 and the Pascal support library coordinate access to files opened via `reset` or `rewrite`. Channels are numbered 0 through 15 (decimal), which means a Pascal program can have up to 16 files opened at one time. If the program is overlaid, however, RT-11 reserves channel 15 as the overlay load channel, which is then unavailable for use by the program. For overlaid programs, up to 15 files may be open at the same time.

Normally, Pascal users do not need to know which channels are associated which files in their program; all file accessing is handled automatically by the Pascal support library. However, in some special applications, the channel assignments may be important. For example, a user wishing to open a file from Pascal and access the file with specialized MACRO-11 subroutines needs to make sure the same channel is accessed from Pascal and from MACRO-11.

As the preceding example program PCHAN.PAS shows, input is associated with channel 16 and output with channel 17. These channels are not true channels because actual I/O transfers are made through `.TTYIN` and `.TTYOUT` system calls, which do not require channels. They are assigned to `input` and `output` so the support library can reference all open files by channel number.

The preceding example program also shows the order in which the Pascal support library allocates channels, from high to low (channel 15 down to channel 0).

If you use `reset` or `rewrite` on the standard files `input` or `output`, or if you allocate another file variable to `TT:` (the console terminal), the support library assigns a channel from its own list of available channels.

**NOTE**

> Because the support library maintains its own list of available channels, it does not use RT-11 system library routines to get the next free channel. Conflicts in channel allocation arise when a Pascal program calls MACRO-11 external modules that open a file on a specific channel and then attempts to open a file on the same channel.

When a file is closed, the channel associated with the file is place on the list of channels available to the program.

## Run-Time Organization

### Form of the Generated Code

Pascal–2 code is divided into program sections called "psects." The psects for the main program and any separately compiled procedures are combined with the Pascal support library by the Linker to produce an executable program image. The use of multiple psects arranged in the order the Linker encounters them provides greater flexibility for the combination of individual procedures into a program. (The "blank" psect is normally placed first.)

The compiler generates these psects:

"blank" The instruction code for the compilation unit. The blank psects for all compilation units are concatenated; compiled code will not attempt to write to this psect.

`CONSTS` Contains all constants generated by the compiler. This includes constants declared by constant declarations or implicit in the code. This psect also contains jump tables generated by case statements, so there is a complete separation of instruction references and data references. The `CONSTS` psects for all compilation units are concatenated; compiled code will not attempt to write to this psect.

`DIAGS` Contains line number and procedure name data used in the printing of the run-time walkback. The information is encoded to save space. This psect is not generated if the **nowalkback** switch is specified in the compilation.

`GLOBAL` Contains all global variables used in the main program and external procedures. This psect is arranged so that the global variables are shared among all procedures. The main program and all procedures that reference global variables should have exactly the same declarations. The size of the resulting psect is that of the largest `GLOBAL` psect generated by any of the compilation units.

If the **own** switch is specified in the compilation, this psect is instead named with the first six characters of the program name, allowing multiple global variable segments. Compiled code will write to this psect.

`P$DYNL` A two-word psect defining a dynamic link to the Post-Mortem Analyzer. (The PMA prints the walkback.) The first word of the psect is a pointer used by the PMA to trace the stack frames for the walkback. With walkback enabled, the second word of this psect contains the address of `P$PMA`, the entry point of the PMA. If the **nowalkback** or **nomain** compilation switch is used, the second word contains a zero and no jump is made to the PMA.

`SHIFTS` Generated only if the target machine does not have the EIS hardware option (**sim**). This psect contains a table of shift instructions that simulate multiple shifts. The psect is overlaid in a manner similar to `TABLES` and is treated as read-only by the compiled code.

`TABLES` Contains bit tables used for access to set elements and individual bits within a word. All Pascal compilations generate this psect, but all copies will be overlaid by the Linker so that only a single copy will exist in the final program. Compiled code will not attempt to write to this psect.

The following table summarizes the attributes of the various psects. Refer to the MACRO-11 manual for further information on the meaning of the attributes.

| Psect Name | Attributes |
|---|---|
| "blank" | RW,I,LCL,REL,CON |
| CONSTS | RW,D,LCL,REL,CON |
| DIAGS | RW,D,LCL,REL,CON |
| GLOBAL | RW,D,GBL,REL,OVR |
| P$DYNL | RW,D,GBL,REL,OVR |
| SHIFTS | RW,I,GBL,REL,OVR |
| TABLES | RW,D,GBL,REL,OVR |

So that Pascal programs may be included in libraries, each Pascal-2 object file has a module name consisting of the first six characters of the output file name. Thus a program compiled with the line:

```
.R PASCAL
*RESPROG = HDR,INPROG
```

will have the module name RESPRO.OBJ. This compilation performs "source concatenation." Note that with source concatenation INPROG.PAS must not contain a **program** statement or compilation errors will result.

## Memory Organization

On the PDP-11 a program has access to 32768 words (frequently abbreviated to 32K). The exact arrangement of storage is determined by the commands to the Linker, but a typical program may look something like Figure 2-1, which represents a snapshot taken during execution. The numbers are representative; actual values vary from program to program.

### RT-11 System Area

The RT-11 System Area occupies the first 256 words of all programs and contains interrupt vectors and status indicators used by RT-11. This area is also used for communication between the Pascal program and other programs linked by chaining.

### Program Code

The program code section contains the instructions for the user program. The size of this section is determined by the amount of user code.

### Global Variables

The global variables section contains the global variables used by the Pascal main program and external procedures. The size is that of the largest global variable section in any compilation unit.

### Constants

The constants section consists of all constants, such as strings or real constants, needed by the program. The section also contains the jump tables for **case** statements. The size of this section is determined by the user code.

### Tables

The tables section, which contains data needed by all Pascal programs, is 40 bytes long.

### Run-Time Library

This section contains routines from the Pascal run-time library used by a program. Only those routines needed by a particular program are loaded here.

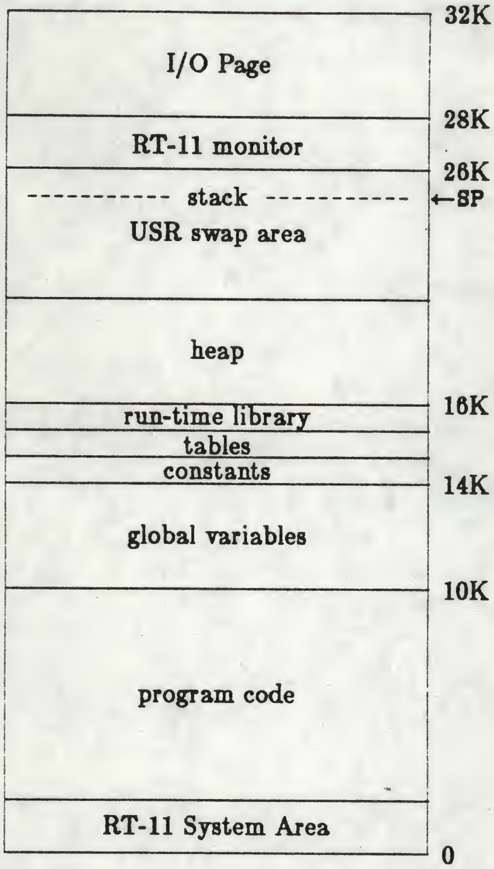


**Figure 2-1. Typical Memory Layout of a Pascal Program.**

### The Stack

The stack contains all variables local to inner blocks of the program, plus parameters, procedure linkage information, and temporary working storage. Upon entry to a procedure or function, space is allocated on the stack (a stack frame) containing space for all storage local to that block. The format of the stack frame is described below.

The stack starts at the highest available address and expands downward, and the heap begins just after the program image and grows upward. This allows the maximum room for growth in both.

The stack pointer (SP) always points to the top of the stack (lowest physical space). If the space available for the stack is too small, the stack pointer will eventually exceed the limits of the stack space and cause the "stack overflow" error.

### The Heap

The heap is an area for dynamically allocated memory used for I/O control blocks, buffers, and variables allocated with **new**.

The heap is allocated from the bottom of available memory and can grow until it meets the stack, which is allocated at the top of available memory.

Space is returned to the heap when files are closed or when variables previously allocated with the standard procedure **new** are deallocated with the standard procedure **dispose**. Such space is then

available for further heap allocation. The error message "not enough memory" results if no space is available to satisfy a request for heap storage.

If a program running under XM is linked as a virtual overlay job, the overlays can be structured in such a way that the excess (and unused) space in the null root is added to the heap's "free list" of available memory. See "The Linker, Overlays, and the Librarian" for details.

For information on ways to monitor the allocation of the stack and heap at run-time, refer to the section on "Monitoring Memory Usage" later in this guide.

### The Monitor and I/O Page

For RT-11 systems other than XM virtual jobs, the monitor space contains the RT-11 resident monitor, the user service routine (USR) if it is set "`NOSWAP`," and any device handlers loaded with the `LOAD` command. The I/O page contains device status and command registers.

For an RT-11 XM virtual job (bit 10 set in the JSW), the monitor and I/O page are not allocated, and the stack will begin at the top of user memory. See "Virtual Jobs and the XM Monitor" for details.

## The Stack Frame

As each procedure or function is entered, space is allocated on the stack for parameters, linkage data, and local use. This space is called a "stack frame"; the "stack" consists of these stack frames.

Figure 2-2 shows the format of a stack frame.

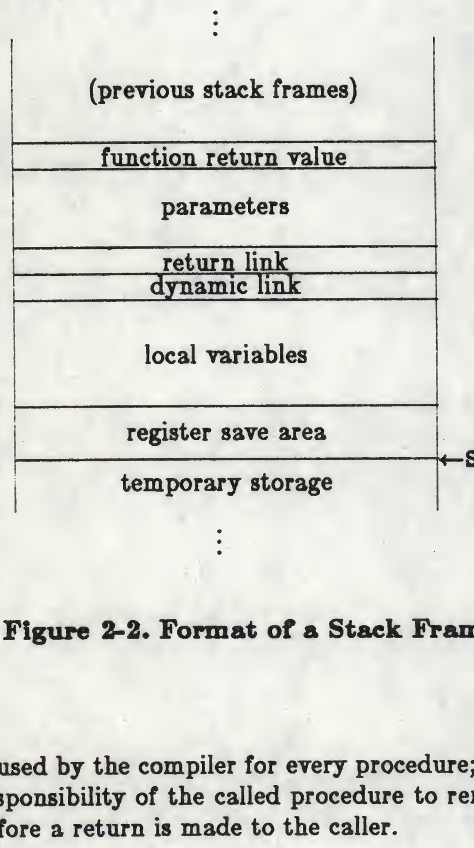


**Figure 2-2. Format of a Stack Frame.**

Not all of the fields will be used by the compiler for every procedure; only the return link is present in every frame. It is the responsibility of the called procedure to remove the parameters and local variables from the stack before a return is made to the caller.

**Function Return Value**

The function return value field appears only for functions. A value assigned to the function name within the function will be stored in this location and left on the top of the stack when the function returns. Space for this field is allocated by the calling routine before evaluation of the arguments for the function call. The space is removed from the stack when the calling routine has no further use for the value.

**Parameters**

The parameter area has an entry for each parameter to the procedure or function. The entry for a value parameter will contain the value of the corresponding argument, while the entry for a variable parameter will contain the address of the argument. Parameters are pushed onto the stack as they are evaluated, in left to right order, so the first parameter to a procedure will be the first one pushed onto the stack.

**Return Link**

The return link is the address to which control will be transferred on return from the procedure or function.

**Dynamic Link**

By default, procedures compiled with **walkback** enabled establish a dynamic link that points to the dynamic link in the previous stack frame. The base of the linked list of stack frames is contained in the first word of the psect P$DYNL. When a run-time error is detected, the dynamic link is used to show the procedure calls that led to the error (the procedure walkback). A dynamic link is not present in the stack frame for procedures compiled with **nowalkback**.

**Local Variables**

The local variable field contains space for all local variables of the procedure or function. The field is allocated upon entry to the block.

**Register Save Area**

This area saves the values of all registers used within the procedure. The registers are saved on entry to the procedure and restored on exit. Only registers actually used are saved. The general registers are stored first, with the highest register used pushed first. (This is important to the algorithm for locating variables in lexically enclosing blocks.)

**Temporary Storage**

In the process of generating code, expressions that are used more than once are computed and the values saved. These values may be saved on the stack if no register is available to hold them. Also, the stack is used to interface with support library routines and the operating system.

## Monitoring Memory Usage

An executable Pascal program is typically arranged in memory with the program code written to low memory, followed by the memory area shared by the stack and heap. The I/O page and monitor are loaded into the high end of memory. (In this discussion, a typical Pascal program uses no Linker options to rearrange memory.) The remaining memory is unallocated and available for the stack and heap.

To arrive at this arrangement, the following events occur:

1. The program code is loaded into memory, with the stack in low memory.
2. The support library initialization procedure is called, initializing the support library and performing the next three steps. (See the "Support Library" section.)
3. The stack is moved to its new location at the highest available memory location with the `.SETTOP -2` directive or at the address specified on the **/M:***n* Linker option.
4. **P$KORE**, the pointer to the top of the heap, is initialized. The heap immediately follows the program code.
5. The user program is started. During execution, if the stack and heap meet, the program terminates with either of two errors, "not enough memory" or "stack overflow," depending on the cause.

Although overlaid programs differ from non-overlaid programs in their arrangement in memory, the same sequence of events occur to load them into memory. In this case, the program code is made up of a root segment and a number of overlay regions into which the progam is divided at link-time. For low memory overlays, if the heap is exhausted, no more memory is available and the program terminates. For extended memory overlays (virtual overlays), the excess memory in the root segment is placed on the heap's free list, extending the memory available for the heap. This and other information on overlays is described in "The Linker, Overlays, and the Librarian" and later in this section.

In certain applications you may find it advantageous to keep track of the stack and heap as they are used by a program. The Pascal support library contains three routines — **space**, **p$inew** and **p$dispose** — that allow you to keep track of memory allocated to the stack and heap. Briefly, the **space** function returns the amount of stack and heap space available to an executing program at a particular time. The **p$inew** function returns the address of a block of memory having a specified length. The **p$dispose** procedure deallocates blocks of memory allocated by **p$inew**. In a later example a boolean function `NewOK` is provided, which not only shows the correct way to use the three routines but also is useful in determining whether enough memory is available to satisfy a request for a block of memory.

Two reasons for monitoring the size of the stack and heap are to find out how close the program is to running out of memory, and to find out whether enough memory is available to perform a given subtask. For example, a checkers-playing program could use these functions (as in `NewOK`) to determine the number of moves that the program can look ahead based on the amount of memory available to perform the look-ahead.

**Space** can be called independent of the other two support library routines, whereas **p$dispose** must be used to deallocate memory allocated by **p$inew**. The routines are described in detail below.

### The 'Space' Function

The **space** function is used to determine the amount of stack and heap space available to an executing program. With this function you can determine how close the program is to running out of memory.

The function **space** must be declared **external** to the program, as shown:

```
function Space: functype; external;
```

where *functype* is the data type of the function and its returned value. The function is usually declared of type **integer** but another data type similar to **integer**, "unsigned" (`0..65535`), can be used to represent the value. The value returned by **space** depends on the amount of space allocated to the stack and heap.

Figure 2-3 is a memory diagram showing a program's arrangement in memory and its allocation of stack and heap space. The diagram also shows the relationship between the stack and heap and the returned value of the **space** function. This diagram is designed as an aid in visualizing the use of the stack and heap and as an aid in understanding the way the value of **space** is arrived at.

As is the case on RT-11, the stack and heap share the same block of memory. The stack begins at the high end of the stack/heap area and grows down; the heap begins at **P$KORE**, the low end of the block and grows up. The **space** function monitors the use of the stack and heap, returning the difference between the top of the stack (**SP**) and the top of the heap (**P$KORE**).

The figure is divided into "times," or snapshots of a program in memory at various stages of its execution. Time 0 is the point at which the user program actually begins execution, after the program code is loaded into memory and the support library is initialized.

The figure contains the following five symbols. Curled braces designate the value that the **space** function would return at the indicated time. Vertical arrows represent stack and heap expansion. **SP** signifies the stack pointer. **P$KORE** is a pointer to the top of the heap. In general, KB stands for K bytes; 64KB stands for 64K bytes or 32K words, the high end of memory; 56KB stands for 56K bytes or 28K words, the end of the RT-11 monitor and beginning of the I/O page; 52KB stands for 52K bytes or 26K words, the bottom of the stack and the beginning of the monitor; 0KB denotes the beginning of memory (low core and vectors).. These memory locations are typical of Pascal programs but may vary from program to program.

Be aware of the fact that the **space** function does not account for the fragmentation of the heap as a result of calls to **new** and **dispose** and the opening and closing of files. Unused portions between the low and high end of the heap are treated as if they are allocated. Again, for extended memory overlays the **space** function does not take into account the memory added to the heap's free list.

See the example under "Example: Function NewOK" for use of **space**.

### Function 'P$inew' and Procedure 'P$dispose'

The function **p$inew** and procedure **p$dispose** are entry points for the standard procedures **new** and **dispose**. **P$inew** allocates a specific sized block of memory, and **p$dispose** deallocates a specific block. As a comparison, the standard procedures **new** and **dispose** determine the size of the block to allocate or deallocate from the length of the data type.

An example use of **p$inew** and **p$dispose** is a cache system in which a program needs to use as much memory as is available for data storage before it writes the data to a file.

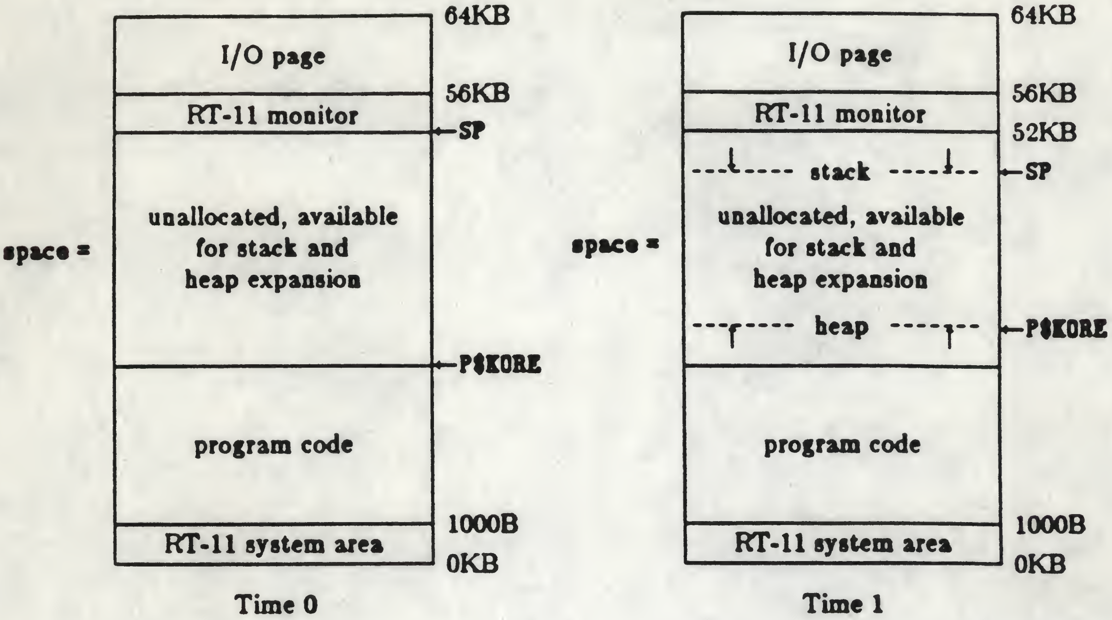


**Figure 2-3. Tracking Memory Usage with the SPACE function.**

*At Time 0, the support library has been initialized but no Pascal statements have been executed. The stack and heap, in sharing the same block of memory, grow toward each other and ideally never meet. If* **space** *is called at Time 0, the value it returns is equal to the size of the stack and heap area (*SP *minus* **P$KORE***). At Time 1, the amount of stack and heap remaining and the value of* **space***, being identical, both decrease by the same amount as stack and heap space is allocated. As time progresses, memory available for the stack and heap continues to be used and disposed of until the program ends or until the allotted memory is exhausted and the program aborts.*

Defined in the support library, these two routines must be declared **external** to the program, as shown:

> **function P$inew**(*blocksize*: *argtype*): *functype*; **external**;
>
> **procedure P$dispose**(*pointer*, *blocksize*: *argtype*); **external**;

where

*blocksize* is the size of the memory block to be allocated or deallocated, in bytes.

*pointer* is the address of block to deallocate.

*argtype* is the data type describing *size* and *pointer*.

*functype* is the data type of the function's return value. The returned value is the address of the block. If there is not enough contiguous memory available to satisfy the request, a value of 0 is returned for numeric data types, **nil** for pointer types. The most common types used with **p$inew** and **p$dispose** are the standard type **integer** and a user-defined type **unsigned**, in the range 0..65535.

For an example showing the use of **p$inew** and **p$dispose** see the code for procedure **NewOK**, below.

**Example: Function 'NewOK'**

The boolean function **NewOK**, provided below as an external, uses the three routines previously described functions to determine whether a block of memory can be allocated, leaving a specified amount of stack space. We recommend that you reserve $200_8$ to $1000_8$ bytes of stack space for parameter and local variable storage and for error processing. The amount of memory you reserve depends on the parameter and local variable requirements of the procedures being called by the program. For instance, if the program calls numerous procedures, each containing a large number of parameters and local variables, the amount of memory to reserve would be greater than for a program that uses smaller procedures.

**NOTE**

> If a program that uses **NewOK** aborts with a "stack overflow" error, the amount of reserved memory is probably not large enough for the amount of stack space required for parameters and local variables. To alleviate this error, increase the amount of memory you are reserving for the stack.

In addition to showing the correct way to use the three routines, **NewOK** can be incorporated into your programs when you need to determine if a request for memory will fail.

**NewOK**, which could be stored in NEWOK.PAS, returns a **true** value if the block can be allocated, **false** if the block cannot be allocated. The first argument to **NewOK** is the size, in bytes, of the memory to be allocated. Use the **size** function to determine the size of a Pascal data type. (The **size** function is described in the Language Specification.) The second argument is the amount of stack space in bytes that you want to remain unallocated, if the block is allocated.

First, the function allocates the desired block of memory with the call to **p$inew**. If **p$inew** returns a zero, this means that there was not enough memory available to allocate a block of that size, and **NewOK** returns **false**. However, just because the block was allocated does not necessarily mean that **NewOK** returns **true**. If the returned value of **space** is less than the amount of stack space you have reserved for variables, etc., **NewOK** is set to **false** because memory would be taken from the reserved block. Of course, the function returns a **true** value if the block was safely available. Finally, the same block of memory is disposed of by **p$dispose**. (Remember, **NewOK** only checks to see if a block of memory could be allocated. To allocate the block, call **new**.)

```
{$nomain}
type
  Unsigned = 0..65535; { Unsigned integer }

function p$inew(Blocksize: Unsigned): Unsigned;
  external;      { Allocate a specific sized block of memory }

procedure p$dispose(Pointer, Blocksize: Unsigned);
  external;      { Dispose of a specific sized block of memory }

function space: Unsigned;
  external;      { Determine amount of stack space left }
```

```
function Newok(Reserved, Stackspace: Unsigned): boolean;  external;
  { Check if block of size "Reserved" can be allocated leaving "Stackspace" }

function NewOk;

  var
    P: Unsigned;  { Address of block if allocated }

  begin  { NewOK }
    P := p$inew(Reserved);              { Try to allocate block }
    if P = 0 then NewOK := false        { No luck }
    else
      begin
        NewOK := space >= Stackspace;   { Check for enough stack left }
        p$dispose(P, Reserved);         { Deallocate the block }
      end;
  end;  { NewOK }
```

To show the correct way to set up the critical variables and call **NewOK**, a checkers-playing program could use **NewOK** to find out how many moves it can look ahead. The sample program provided below does just that. For illustrative purposes, the program, CHECKT.PAS, simulates a real checkers program, making use of a 10,000-word array **Dum** to produce a larger task size. CHECKT uses a linked list to store the look-ahead moves a normal checkers program would make. The program merely illustrates the use of **NewOK** to perform the look-ahead.

```
program CheckTest;

const
  Reserved = 200; { amount of stack space to reserve }

type
  Unsigned = 0..65535;
  Ptr = ^Node;                 { Pointers into search tree }
  Node = record                { Node in search tree }
    Father: Ptr;               { Father of this node }
    Son: Ptr;                  { Pointer to best son }
    Brother: Ptr;              { Link to next brother }
    Value: integer;            { Value of this board position }
    Move: integer;             { Move descriptor to reach node }
    Jump1, Jump2: integer;     { Jumped pieces removed by move }
    Mobility: integer;         { Mobil and deny }
    Attack: integer;           { Pin and threat }
    Gradient: integer;         { Target gradient }
    Bits: integer;             { Scoring bits }
    end;

var
  Alloc: boolean;
  P: Ptr;
  Movesize: unsigned;
  NumMoves: unsigned;                 { number of moves }
  Nextmove: node;
  Dum: array [1..10000] of integer;  { Dummy array simulating a long program }
```

```
function NewOK(Reserved, Stackspace: Unsigned): boolean;
  external;
  { Check if block of size "Reserved" can be allocated leaving "Stackspace" }

begin  { CheckTest }
  Movesize := size(Node);
  writeln('The size of a move is: ',Movesize:1);
  new(P);
  NumMoves := 1;
  repeat
    alloc := Newok(Movesize,Reserved);
    if alloc then begin
      new(P^.son);
      P := P^.son;
      NumMoves := NumMoves + 1
      end;
  until not alloc;
  write('The number of moves necessary to fill up the heap is: ');
  writeln(NumMoves:1);
end.  { CheckTest }
```

The compilation process for this program is as follows:

```
.R PASCAL
*NEWOK
.R PASCAL
*CHECKT

.LINK CHECKT=CHECKT,NEWOK,SY:PASCAL
.RUN CHECKT
The size of a move is: 22
The number of moves necessary to fill up the heap is: 1141
```

## Storage Allocation

The compiler assigns storage for variables of pre-declared types as shown in this table:

| Type | Size (bytes) | Alignment (bytes) |
|---|---|---|
| Boolean | 1 | 1 |
| Char | 1 | 1 |
| Integer | 2 | 2 |
| Real | 4 | 2 ($double off) |
| Real | 8 | 2 ($double on) |
| Text | 2 | 2 |

Space for user-defined types is allocated as follows:

**Enumeration**
If the type has up to 256 members, it is allocated one byte aligned on a byte boundary. If it has more than 256 members, it is allocated two bytes, aligned on a two-byte boundary.

**Subrange**
Allocated in the same way as the parent type.

**Pointer** Allocated two bytes, aligned on a two-byte boundary.

**Array** Allocated the amount of space needed to hold the number of elements specified, aligned in the same way as the element type. The elements are placed in ascending memory locations.

**Set** Allocated one bit for each member of the base type, with the total size rounded up to the next larger full byte. Bit allocation begins with the least significant bit of the first byte. If the size is a single byte, it is aligned on a byte boundary; otherwise it is aligned on a two-byte boundary. A base type that is a subrange is expanded to the full range of possible values before the set is allocated. For example:

```
type
  Color = (Red, Orange, Yellow, Green, Blue);
  Hot = Red..Yellow;

  Colorset = set of Color;
  Hotset = set of Hot;
```

In this example, `Hotset` is allocated the same amount of space as `Colorset`. A maximum of 256 members is allowed; a base type of `integer`, or any integer subrange, has members from 0 to 255.

**Record** Each field in the record is allocated space in the same way as a variable of the same type, in the order specified. The alignment of the record is the maximum of the alignments of its fields.

**Packed Array**
The number of bits needed to contain each element is computed. For example, the subrange type `0..3` requires two bits to contain a value. If the space required for an element is less than a word, the element size is increased to the smallest power of two bits (1, 2, 4, 8, 16) that will contain the value. The array is allocated space to hold the number of elements specified, where each element is considered to be of the size just computed. If elements are allocated eight bits or less, the array is aligned on a byte boundary. If the elements require a word or more, space is allocated as for a normal array type.

**Packed Set**

The same as unpacked sets, except that the size is not rounded up to an even byte and the alignment is to a byte boundary.

**Packed Record**

Each field in the record is allocated exactly the number of bits required to contain it, except that a field of a simple type that would span or cross a word boundary will be forced to begin at a word boundary. Fields are allocated in the order declared, beginning at bit zero (least significant bit).

**NOTE**

The predefined functions **size** and **bitsize** will report the amount of storage allocated to any user-defined structured type. See the Language Specification for details.

## Run-Time Error Reporting

The Pascal–2 run-time error reporting system is intended to simplify error analysis by reporting run-time errors in terms of source lines and procedure names. Upon detecting a run-time error, the reporting system prints a short description of the error, then traces the execution history, procedure by procedure, from the point of error back to the main program. This is called a "walkback," or "traceback."

Errors are detected by the hardware or by special checks inserted in the generated code. After an error is detected, control of the program then passes to an error routine, which closes all open files and prints an error message and stack traceback.

The walkback consists of the following:

- The message header, which includes the program name, type of error, and program counter at the time of the error. The two types of run-time errors are "fatal" and "I/O." Fatal errors are unrecoverable; I/O errors are recoverable. For details on I/O-error recovery, see "I/O Error Trapping" later in this section. The program counter is the location at which the error occurred. If you utilize the walkback, the program counter is of little use to you since the location of the error is given as a line number in a procedure.
- A description of the error. See Appendix B of this guide for a detailed explanation of the error messages. By modifying OPERRO.PAS, you can change the wording of Pascal run-time messages if you so desire. See the section on "Customizing Error Reporting" for details.
- For I/O errors, the error code and the file name of the file causing the error. The error code is printed in both decimal and octal. On RT-11, all I/O error codes are errors detected by the support library and are described in Appendix B of this guide. The file name includes such information as the device name, file name and extension.
- The location of the error in terms of line number and procedure name. The line number refers to lines in the overall source program, not statements in individual procedures. (For external procedures, the line number refers to lines in the external module.) A special case arises when a run-time error is detected in an external procedure compiled with **nowalkback**. In this case, the location of the error is given as an octal address.
- The reverse sequence of active procedure calls back to the main program, if the error occurred at a level other than the main program.

The walkback can be disabled at compile time; to do this, use the **nowalkback** compilation switch. When **nowalkback** is selected, the message header and the error message are printed but not the walkback.

**Examples**

Example 1.

This example provides a look at a possible run-time error condition and the resulting error walkback.

```
.RUN PACKER

PASCAL--Fatal error at user PC= 2316B
Array subscript out of bounds

Error occurred at line 140 in procedure arrangetree
Last called from line 326 in procedure switchnodes
Last called from line 402 in procedure getdep
Last called from line 579 in program packer
```

Example 2.

A procedure called recursively may have many consecutive activations. In this case, the number of identical lines is indicated by the note `(nn times)` after the location description. Appearing below is the walkback of a program that looped recursively until the stack overflowed.

```
.RUN EXTRA

PASCAL--Fatal error at user PC= 5223B
Stack overflow

Error occurred at line 124 in procedure walk
Last called from line 290 in procedure reanalysis (898 times)
Last called from line 423 in procedure unloadbits
Last called from line 440 in procedure matrixmask
Last called from line 535 in procedure processleftop
Last called from line 608 in program extra
```

Example 3.

This example illustrates the walkback produced as a result of an I/O error that is not trapped by the user.

```
.RUN REFORM
File to reformat: LSAT.TXT

PASCAL--I/O error at user PC= 2166B
Can't open file
I/O error code= 11. (13B) in file: DK:LSAT.TXT

Error occurred at line 44 in procedure openfile
Last called from line 69 in program reformatter
```

Example 4.

This example shows the walkback produced as a result of a run-time error in an external procedure compiled with **nowalkback**. Note that if the external procedure had been compiled with **walkback** (the default), the location of the error would be in source terms.

```
.RUN DIFFS

PASCAL--Fatal error at user PC= 1336B
Division by zero

Error occurred at location    1336
Last called from line 32 in program diffs
```

## I/O Error Trapping

Pascal–2 permits you to write programs that trap and detect many kinds of I/O-related errors that normally would be fatal. Three predefined routines — procedure **noioerror** and functions **ioerror** and **iostatus** — facilitate this trapping of I/O errors. Using these routines, you have the ability to process I/O errors with your own code. You have two options: terminate the program at the occurrence of an I/O error (you can print your own diagnostics), or continue execution in spite of the error. The choice depends on the need.

Since these three routines are predefined in the compiler, they do not need to be declared in your program. They accept a file variable as their only parameter. Details are supplied below.

procedure **noioerror**:
: Specifies that the calling program will handle any I/O errors that result from reading or writing to the specified file. The file must be open before **noioerror** is called. This procedure performs the same function as the **/go** file control switch on **reset** and **rewrite** statements.

function **ioerror**:
: Determines the status of the last I/O operation that the program performed on the specified file. This **boolean** function returns a **true** value if an I/O error has occurred, **false** if the operation was successful.

function **iostatus**:
: Returns the integer error code that describes the last attempt to access a file. This function helps your program determine the cause of the error. Your program can either bypass the problem and continue processing, or terminate so you can correct the problem. The I/O error is detected by the Pascal–2 support library. Pascal–2 error codes, along with the text of the error message and a brief explanation of the cause, are listed in Appendix B of this guide.

When you call these routines, you are responsible for checking the status of each I/O operation, to ensure that it was successful. If you fail to check the status and an error occurred, the results are unpredictable.

The following program illustrates the use of these procedures. The program is designed to continue executing despite an I/O error. The call to **noioerror** indicates to the run-time system that errors

detected on the standard file **input** will be handled by the program.

```
program Iotest;

  var
    I, Times: integer;

  begin
    Noioerror(input);
    for Times := 1 to 4 do
      begin
        write('Type an integer: ');
        read(I);
        if Ioerror(input)
          then writeln('Error detected.  Status=', Iostatus(input))
        else writeln('The integer was: ', I: 1);
        readln;
        writeln;
      end;
  end.
```

If this program is compiled and run, the following results might be produced. The first entry results in a successful read of the integer I. The second and third entries result in a Pascal–2 run-time error. See Appendix B for a list of run-time error messages and associated numbers. The final entry is successfully read, and the program ends. (Under normal conditions, the first error would cause the program to abort.)

```
.RUN IOTEST
Type an integer: 1234
The integer was: 1234

Type an integer: 123456789
Error detected.  Status=     23

Type an integer: FFG
Error detected.  Status=     23

Type an integer: 77
The integer was: 77
```

The I/O error-trapping procedures can be used to determine the reason that a file could not be opened. To use this feature, specify the fourth parameter on calls to **reset** and **rewrite**. Specifying this fourth parameter keeps the **reset** or **rewrite** from trapping a normally fatal "open" error. This allows your program to recover and continue, or terminate under your control.

The following program illustrates the use of **ioerror** and **reset/rewrite**. The program attempts to open a file called TEST.DAT on the device **XXXX:**, a fictitious device name. The error is detected

by Ioerror.

```
program Opnerr;

  var
    F: text;
    Status: integer;

  begin
    reset(F, 'XXXX:', 'test.dat', Status);
    if Ioerror(F) then
      writeln('I/O status=', Iostatus(F));
  end.
```

When this program is compiled and executed on RT-11, it yields the following output. The value 11 is the support library run-time error code for "can't open file."

```
.RUN OPNERR
I/O status=      11
```

**Customizing Error Reporting**

The flexibility of the Pascal–2 run-time organization allows you to not only handle I/O errors in your code but to customize the run-time error diagnostics to suit your needs. Included in the distribution kit are two Pascal source files, OPERRO.PAS and UERROR.PAS, which let you modify the way in which errors are reported. The object-file equivalents of these two procedures are in the Pascal support library. The changes you make can be used for a one-time debugging run, or they can be permanently installed in the support library.

OPERRO.PAS contains the entry point **P$ERROR**, which the support library's error-handling routine calls to print the message header and text of the run-time error message. This source file is provided so you can change the wording of any error message simply by editing the source.

UERROR.PAS contains the entry point **P$UERROR**, which is called following **P$ERROR** to print additional information about the error. This procedure contains two boolean constants set to **false** in the release version, each controlling (inhibiting) the printing of a separate set of diagnostics. Initially, with the booleans set to **false** nothing is printed. But by editing UERROR.PAS and setting one or more constants to **true**, you can receive a file dump of the offending file and/or a memory map of the program. The **const** fragment below shows the two constants.

```
const
  Dump_Memory = false; { Print a memory map }
  Dump_File = false; { Print detailed file dump }
```

By using UERROR.PAS, you can add your own code to UERROR.PAS for the printing of more specialized debugging information, and you can print out the values of critical variables used in the program. To print variables you must include the program's global variable declarations in UERROR.PAS. The ability to print critical variables is useful when you have a program with many overlays or when the program is too large to run with the Debugger.

When a run-time error is detected, several steps are taken:

1. Control of the program transfers to the **$ERR** error-control module, in the Pascal support library. **$ERR** collects information about the error from the library data area.
2. **$ERR** calls **P$ERROR** (in OPERRO.PAS) to print the message header followed by the error message.

3. **$ERR** then calls **P$UERROR** (in UERROR. PAS), which does nothing (by default) but can be modified to print a memory map of your program and/or dump the contents of the offending file.

4. On return from **P$UERROR**, the error-control routine **$ERR** transfers control to the Post-Mortem Analyzer (PMA), which prints the error walkback, and the program terminates.

When you use a modified version of one or both of these procedures as externals, you need not declare them explicitly in the main program. After the altered version(s) of these procedures are compiled (with **nomain**), simply specify the module name(s) on the Linker command line after the program name. The Linker substitutes your version for the version in the support library. This sequence of commands should be used:

```
.R PASCAL
*PROG
.R PASCAL
*UERROR
.R PASCAL
*OPERRO

.R LINK
*PROG=PROG,UERROR,OPERRO,SY:PASCAL
```

Another way to override the version in the support library is to include the modified OPERRO.PAS and/or UERROR.PAS as part of the main program. The **%include** directive does this easily. (See "Implementation Notes" in this guide for use of **%include**.)

For example:

```
%include 'operro';
%include 'uerror';
```

When using the **%include** directive, compile and link the main program as you normally would. The Linker resolves the references to **P$ERROR** and **P$UERROR** with the procedures included in the program. The program **PROG**, above, would be linked in this manner with these commands:

```
.R PASCAL
*PROG

.R LINK
*PROG=PROG,SY:PASCAL
```

The constant **Dump_Memory**, if set to **true**, causes the program to print a memory dump, or map, of the program showing the program's use of memory. The map is printed by the external procedure **memmap**, a Pascal support library routine. Although **memmap** is declared in UERROR.PAS, you can use it with any Pascal program, independent of UERROR.PAS. Simply declare it in the program as an external with no parameters. The map helps you determine the way dynamic memory is being

allocated and perhaps the reason your program is running out of memory.

```
.RUN DIAL

PASCAL--Fatal error at user PC= 1276B
Attempted reference through NIL pointer


Memory Map:

Pointer to top of stack - RESR6 : 140060B
Pointer to global data -  RESR5 : 13102B
Standard Output :13076B
Standard Input :13100B
Saved Output^ :33256B
Saved Input^ :33316B
Last file access :
               File @ 33256B TI :       .
Active files :
               File @ 33316B TI :       .
               File @ 33256B TI :       .
PMA active
No free list
No orphan list
Last User PC 5116B
Pointer to top of heap (P$KORE) : 33556B

Error occurred at line 29 in program dialphones
```

The constant `Dump_File` can be used to print a detailed dump of the Pascal and RT-11 file structures when an I/O error is detected. When `Dump_File` is set to `true`, the support library module `fdump` is called to dump information about the file. Programmers familiar with the structure of the file variable may find this information useful in diagnosing obscure file problems.

The following listing is the file dump produced by `fdump`. Although you wouldn't want the file dump printed at each occurrence of a run-time I/O error, in certain circumstances the file dump may help you diagnose more obscure errors. Lines such as "buffer size" display the information first in octal,

denoted by the B, then in decimal.

```
.RUN COVER

PASCAL--I/O error at user PC= 1072B
File is not a random access file. Use /SEEK
I/O error code= 27. (33B) in file: DK:LSAT.DAT

File information for file variable at:34144B
Contents of file variable:
Ptr: 34204B Pointer to data in file buffer
Stat:250B File status
                    Records are Blocked
                    Sequential File
                    Read operations permitted
                    Pending End of File
                    Permanent File
                    Non interactive device/.read
                    Text file
                    Current character not defined
Name: DK :LSAT  .DAT
File Size in Blocks: 1B 1.
Channel : 15
Current Block : 0B 0.
Buffer Address : 34204B 14468.
Buffer Size : 1000B 512.
Handler : RK:
Device Info :
                   Random Access Device
Record Size in bytes : 1B 1.
Pointer to End of Valid Data : 34204B 14468.
Records Per Block : 1000B 512.
Terminal # : 0
Last I/O Error : 27

Error occurred at line 10 in program cover
```

## Error Termination Status

Both the Pascal-2 compiler and Pascal programs return a termination status when they exit. The Pascal-2 compiler terminates with a "severe error" status if it detects compilation errors. Upon detecting an error while running, such as "subscript out of bounds," a Pascal program also will terminate with a "severe error" status. Otherwise, a "successful completion" status is returned.

The termination status can be used by the command file processor and the batch processor to terminate a command stream that encounters an error. For instance, a command file that compiles and links a Pascal program can use the compiler termination status to detect any errors and skip the link step.

The **Exitst** procedure is a support library routine that sets the termination status of an executing program when a "severe error" status is detected. The procedure's integer argument determines the termination status for any program that calls it. When a "severe error" status of 4 is passed, the procedure also invokes the post-mortem analyzer to create a walkback of the program execution from the point of failure. **Exitst** takes its integer argument, as shown:

```
procedure Exitst(Status: integer);      { procedure declaration }
  external;
```

Call the procedure at a point in the program where you want to exit in case of a severe error, as shown:

```
begin       { program Severe }
  :
  Exitst(4); ------------------------- terminate with severe status
  :
end.       { program Severe }
```

A status of 1 means normal termination; any other status means that an error terminated the program.

## Implementation Notes

### Multiple Source Files

To combine multiple Pascal-2 files into a single compilation unit, you may use multiple input files on the compilation command line, the **%include** extended language feature within the program text, or both.

The choice depends on the need. If, for instance, you are preparing programs for different machines, you can separate machine-dependent data from your individual programs and use the configuration data in a "header" file on the compilation command line.

The **%include** directive allows the inclusion of separate text files within a program, thus simplifying the calling of external procedures. The directive is written as:

**%include '*file-name-string*';**

The contents of the file specified by ***file-name-string*** are inserted at the point of the **%include** directive. The string must contain at least the name of the file; if no file extension is specified, .PAS is assumed. In addition to the file name and extension, ***file-name-string*** can contain such information as the logical device name and disk volume number of the file.

The single quotes ('...') enclosing ***file-name-string*** are optional and provide portability to other implementations of Pascal-2. Despite their optional nature, we recommend that you use the single-quote delimiters on all **%include** directives.

Examples:

```
%include 'hdr';
%include 'sy:string';
%include 'dk0:libdef.pas';
```

Each included file may itself contain **%include** directives, to a maximum nesting of seven levels.

The example below illustrates the use of both header files and the **%include** directive.

Assume that the source file CONFIG consists of this:

```
{ This file contains configuration data that is }
{ subject to change from installation to installation. }

const
  MaxEntries  = 10;       {entries allowed}
  Debug = false;          {if true, make debugging calls}
```

Assume also that the source file COMDEF consists of this:

```
{ This file contains the definitions of some external }
{ procedures, together with the type declarations needed }
{ by the main program and the external routines. }

const
  NameSize = 24;          {size of name field}

type
  DataItem = record       {describes a customer}
    Name: packed array [1..NameSize] of char;
    Age: 0..maxint
    end;
```

```
procedure ReadData(var ThisItem: DataItem; {result of read}
                   var Done: boolean {No more items} );
  external;

procedure WriteData(ThisItem: DataItem {item to write} );
  external;
```

And assume that the source file EXAMPL consists of this:

```
%include 'comdef';

var
  Base: array [1..MaxEntries] of DataItem;
  Buf: DataItem;

  Counter: 0..MaxEntries; {count of items in data base}
  I: 0..MaxEntries;       {induction var}

  Done: boolean;          {set when no more items}

begin
  Counter := 0;
  repeat
    ReadData(Buf, Done);
    if not Done then begin
      Counter := Counter + 1;
      Base[Counter] := Buf;
      end;
  until Done;

  { Process data base }

  for I := 1 to Counter do
    WriteData(Base[I]);
end.
```

These files are compiled with the command:

```
.R PASCAL
*CONFIG,EXAMPL
```

The result is an object module, EXAMPL.OBJ, containing the output from the compilation of CONFIG, COMDEF, and EXAMPL, concatenated in that order. The object module can then be processed through the Linker to produce an executable image.

Any compilation switches used will apply to all input files.

### Local Files Closed on Procedure Exit

Consider a procedure (or function) that opens one or more files local to that procedure. Assume that the file variable for the file is defined local to that procedure. When the procedure exits and returns to the calling routine, all files defined local to that procedure are closed. This convention is necessary because upon procedure exit all local variables are deallocated. Once local variables are deallocated, they cannot be referenced again. Therefore, if a local file variable is deallocated, your program can no longer access that file, and no other program may access the file until your program terminates.

To prevent files opened in a procedure from being closed upon procedure exit, define the file variable as a global variable. Since the file variable is in the global data area, the file remains open and accessible until the file is explicitly closed or the program terminates.

### Specifying the Location of the Compiler's Work Files

The Pascal–2 compiler opens several temporary scratch files when it compiles a program. For large Pascal programs these files can become quite large (several hundred blocks) and they can be used quite heavily. The V2.1 compiler will attempt to open its scratch files on the logical device called **WK:**. If this logical device does not exist, the scratch files are opened on **SY:**, the system device.

If you are running on a multi-disk system, and the disk you are using has very little free space, you can assign **WK:** to some other disk that has more room. Use the **ASSIGN** command to do this, as shown below:

```
.ASSIGN DL1 WK
```

This command associates the disk DL1: with the logical name **WK:**. The compiler then opens its scratch files on DL1:.

If your system has different kinds of disks, you should assign **WK:** to the fastest disk on your system. This will reduce compilation time for large programs. You can experiment by using the **times** compilation switch to see if there is a significant change in compilation times with various disks on your system.

### Variable Initialization

The Pascal standard states that variables must be initialized before they are used. Otherwise, their values are unpredictable. Pascal–2 catches most uninitialized variables but can't possibly flag all of them. In short, variable initialization is the programmer's responsibility.

Obvious cases are easily detected, but more complex violations such as the initialization of I below are not caught by the compiler.

```
var
  I, X: integer;

begin
  read(X);
  if X <= 0
    then I := 0     ------------------------  variable is initialized here
  else I := I + 1;  ------------------------  but not here
end.
```

### Reading Command Lines

To prompt for and read the command line from an interactive program, simply write the prompt and read the command line, as shown in the following simple example:

```
var
  CommandLine: packed array [1..80] of char;

  begin
    write('*');
    readln(CommandLine);
    writeln(CommandLine);
  end.
```

However, if you execute the above program as a batch job or an indirect command file, the RT-11 batch processor still expects the command line to be entered from the terminal. This can defeat the whole purpose of running batch or indirect command files; you may want to be free to do other things while the batch job is executing. In this situation, the command line can just as easily be contained in the batch or command file.

With the use of the support library procedure **getlin**, your program can read the command line from the batch processor as well as from the terminal. This flexibility allows you to write programs that can be executed in batch mode or interactively using the same method of obtaining the command line.

The entry point **getlin** is defined in the support library in OPTRAP.MAC, a collection of run-time trap routines and entry points that must be in the root segment of any Pascal program that uses overlays. **Getlin** is declared as an external procedure.

The call to **getlin** involves two entry points. The first entry point is **getlin** itself, which simply jumps to the second entry point, **p$gtln**. **P$gtln**, which is contained in the support library module OPGTLN.MAC, then writes the '*' prompt and reads the command line. From **p$gtln**, control returns to the module that originally called **getlin**.

The use of dual entry points to read a command line provides the ability to overlay the actual **p$gtln** code against the calling program without overlay conflicts. In this way you can process the command line in the first overlay segment, with the rest of the program overlaid in successive segments.

## NOTE

> Though **p$gtln** is a separate entry point callable from **getlin**, we recommend that you avoid calling **p$gtln** directly. The results can be unpredictable.

**Getlin** and its required parameters and data types must be declared in your program similar to the following:

```
type
  CmdIndex = 1..CmdLen;
  CmdBuffer = packed array [CmdIndex] of char;

var
  Cbuff: CmdBuffer;
  Cblen: CmdIndex;

procedure getlin(var Cbuff: CmdBuffer;  { resulting line }
                 var Cblen: CmdIndex);  { length of line }
  external;
```

where

*CmdLen* is the maximum length of the command line. The length of the command line should be at least 80 characters to allow for any command line up to that length.

*Cbuff* is the packed array of characters into which the command line is read. Spaces may be used in the command line but are stripped off during the read.

*Cblen* is the length of the command line in bytes. If *Cblen* is returned zero, a command line is not present.

The support library passes the parameters to **getlin** through a temporary storage area, **P$AREA**. In this way both local and global variables can be passed to **getlin** without restriction. In addition

to protecting the command line from being trashed, P$AREA alleviates memory overwrite problems caused by the USR's swapping in and out of memory to fulfill a request.

The following sample program, FPRINT.PAS, shows a way to check for and read a file name specified in a command line. After it has read the file name, the program simply prints the contents of the file.

```
program FilePrint;

  const
    CmdLen = 80;

  type
    CmdIndex = 1..CmdLen;
    CmdBuffer = packed array [CmdIndex] of char;

  var
    Cbuff: CmdBuffer;
    Cblen: CmdIndex;
    Filename: packed array [1..CmdLen] of char;
    Ch: char;
    I: integer;

  procedure getlin(var Cbuff: CmdBuffer;  { resulting line }
                   var Cblen: CmdIndex);  { length of line }
    external;

  begin
    getlin(Cbuff, Cblen);             { make command line available }
    if Cblen <> 0 then begin          { command line present }
      for I := 1 to Cblen do
        Filename[I] := Cbuff[I];
      reset(input,Filename);          { open the input file }
      while not eof do begin          { once per line }
        while not eoln do begin       { once per char }
          read(Ch); write(Ch);
          end;
        readln; writeln;
        end;
      end
    else writeln('Err -- No file name given');
end.
```

Compile and link the above program with these steps:

```
.R PASCAL
*FPRINT
.LINK FPRINT,SY:PASCAL
```

The program then can be executed interactively or indirectly with the same results. Below, the

the contents of CODE.DAT is directed to the terminal.

```
.RUN FPRINT  ------------------------------------------ interactive execution
*CODE.DAT
   :
 { contents of CODE.DAT }
   :
```

For indirect execution, place the command and command line in a separate file with a .COM extension. FPRINT.COM could contain the following lines:

```
RUN FPRINT  ------------------------------------------ the command
CODE.DAT  ------------------------------------------ the command line
```

The at-sign indirect command symbol is used to execute the command file.

```
.@ FPRINT
.RUN FPRINT
*CODE.DAT
   :
 { contents of CODE.DAT }
   :
```

**NOTE**

A possible drawback to the use of **getlin** is slower program execution, especially on floppy-disk systems that allow the USR to swap.

### Foreground Operation

For foreground operation, allocate additional memory to ensure sufficient space for the stack, the heap, and file buffers. Each Pascal file requires about 300 words (more for large buffers), so allocate at least 600 words for the default input and output files. Use the **/BUFFER:** switch, as shown below. The period after 1024 signifies an octal value.

```
.FRUN <file-name>/BUFFER:1024.
```

### Lazy I/O

For standard Pascal, an interactive input file, such as terminal input, poses a problem. A program must always be able to determine the current status of an open file, i.e., it must be able to retrieve the current record from the buffer variable (F^) and the current values of eoln and eof. Since an interactive file is being created as it is being read, the program must periodically wait for you to enter a full line before it can determine the "end of line" (or the "end of file") and keep the file well-defined. In this way the program is not synchronized with interactive input.

Pascal-2 uses an input interface known as "lazy I/O" to handle input from text files. Since a file's status needs to be defined only when the program actually refers to it, lazy I/O can safely delay any input operation until the program uses its results. When a Pascal program requests an input operation on a text file, the operation is recorded for later use. The delayed operation is triggered by any subsequent reference to the file's buffer variable, the eof or the eoln value. The delay is invisible to the program but is visible to the user from the way the program is synchronized with interactive input.

To use lazy I/O, you need to be aware of its effect on synchronization of input and output operations. As an example, consider a simple program that reads its standard file input, which is connected to a terminal. The program prompts for each line, and stops at the end of the file. The design of the program is dictated by two requirements:

1. For the prompt to be effective, it must appear before the user is required to type the line.
2. To detect the end of the file correctly, the program must check for it before reading each line.

To meet both of these requirements, the program must print the prompt before an operation is performed that requires the next line of the file to be known: checking for "end of file" or reading the line. The following example shows this design.

```
program Interactive(input, output);

begin
  write('prompt:');
  while not eof do
  begin
    readln;
    write('prompt:');
  end
end.
```

**Incorporating Lazy I/O into V2.0 Programs**

Since lazy I/O changes the way Pascal–2 performs interactive I/O, programs compiled with Pascal–2 Version 2.0 may have to be revised so they conform to the new scheme. For example, the 2.0 version of the above program might have been coded as follows:

```
program Interactive(input, output);

  begin
    while not eof do
      begin
        write('prompt:');
        readln;
      end
  end.
```

If you compile this program and execute it, the program appears to "hang." In reality, the program is waiting for input; it is attempting to define the value of **eof** in the **while** statement, before it has the chance to write the prompt. Unfortunately, you will never see the prompt until you type in a line. This means that you will be prompted for the line you have just entered. In this case the program is "out of sync" with input from the terminal.

## Terminal I/O

The RT-11 terminal driver acts as an intermediary between a Pascal program and the video terminal from which it is running. The terminal driver, under operating system direction, collects characters one at a time and at end of line, sends a full line of text to the screen (for a **writeln**) or to the executing program (for a **readln**).

For example, when a **read** statement reads input from a terminal, the terminal driver reads each character and places it in an internal buffer until the terminal driver encounters an end of line (a carriage return, line feed, escape or Control-Z ^Z). At this point, the terminal driver sends the entire line to the program and lets the program process the line one character at a time.

The opposite is true for **write** statements, where no buffering is performed and each character is displayed on the terminal at the time the program writes it.

As previously mentioned, the **writeln** statement directs the terminal driver to write the rest of the current line and prepare for the next line of output. The usual output sequence sent by the terminal driver is: data, carriage return, line feed. This sequence prints the data, returns the cursor to the first position of the newly printed line and moves the cursor to the beginning of the next line.

The normal sequence of commands issued by the terminal driver may not particularly suit special I/O applications such as direct cursor addressing or special function key processing. To gain control of the terminal-I/O command sequence, use the /odt file control switch in conjunction with the **/buffersize:n** switch on the **reset** statements. Used together, the /odt and **/buffersize:1** switches allow your program to read terminal input one character at a time without the need for a line terminator. To recognize special-function keys such as the **PF1** ("gold") key on VT100 keyboards, use the **readsn** procedure available in the Pascal support library. Both of these features are described in the following sections.

### Single-Character or 'ODT' Mode

On RT-11, single-character input between a video terminal and an executing Pascal program is accomplished with the use of the **/odt** and **/buffersize:1** file control switches on the **reset** statement that opens `TI:` as the input file (treated as a **text** file).

When the **/odt** file control switch is specified, each character read from the keyboard processed immediately without the program waiting for a carriage return or other action character. The **/buffersize:1** switch sets the terminal's buffer size to 1 and places the device in single-character mode.

Example:

```
reset(input,'ti:/odt/buffersize:1');
```

A **reset** statement having these switches creates a new file control block and buffer and redirects all references to standard input to the terminal, known to the support library as `TI:`. Consequently, the library calls the system routine `.TTYIN` rather than `.READ` to perform the terminal I/O. **Reset** overhead is minimal because no actual `.LOOKUP` is performed to open the file.

In single-character or "odt" mode, a program reads input from the terminal one character at a time without waiting for a line terminator (i.e., the line-feed and escape characters). As a result, the programmer is responsible for echoing each character as it is read.

**NOTE**

> Though not a line terminator *per se*, the **RETURN** key terminates a line of input because it performs a carriage return then a line feed, which terminates the line. The null and carriage-return characters are ignored.

When line-feed and escape characters are encountered, the **eoln** flag is set to **true** and **input^** points to a space. Likewise, detection of Control-Z (^Z) or the physical end of file sets **eof** to **true** and **input^** to a space.

```
program Single;
  var
    Ch: char;

  begin  { Single }
    reset(input,'ti:/odt/buffersize:1');
    repeat  { Forever -- end it with Control-C }
      write('Prompt >');
      while not eoln do begin
        read(Ch);
        write(Ch);
        end;    { while }
      readln;
      writeln;
    until false;
  end.  { Single }
```

Program **Single** prompts for input from the terminal. The characters you type appear to be echoed to the screen normally as they are typed. However, the program, and not the operating system, is doing the echoing (the **write(Ch)** statement).

**Function 'ReadSn'**

A second method for reading a single character from the terminal (a **text** file) entails the use of the Pascal support library function **readsn**. In addition to reading a single character at a time, **readsn** allows your program to recognize and correctly interpret the special function keys that are available on many terminals.

**Readsn** has the same effect as "odt" mode with a one-byte buffer size and no character echo. **Readsn** does none of the normal text-file processing done by the support library, in which escape characters (<ESC>) are stripped from the file. Since **readsn** does not require a special buffer or file control variable, **reads** and **readlns** from standard input still produce the expected results. You must, however, buffer the characters internally in your program if you wish to save the characters that are typed.

Declare the function as an **external**, as shown:

```
function ReadSn: char;  external;
```

The returned value of **readsn** is the current character read from the terminal.

Special-function keys send a sequence of characters that is interpreted as one key. An example of special-function keys is the top row of keys on the VT100's 10-key keypad marked **PF1** to **PF4**. With the use of **readsn**, the sequence of characters making up special-function keys can be picked off one character at a time. For example, the **PF1** key is made up of the three-character sequence '<ESC>OP' (<ESC>, O and P). By reading the sequence one character at a time using **readsn**, a program can determine that the **PF1** key was pressed and not a carriage-return or line-feed key followed by 'O' then 'P.'

## Random Access to 'Text' Files

The **seek** procedure cannot compute the location of a particular record (line) within a file of type **text** because the lines are of variable lengths. The Pascal support library supplies two external procedures, **getpos** and **setpos**, that simulate random access to **text** files.

These two procedures are not predefined and must be declared in your program as **external**. **Getpos** determines the starting location of the next line of a file, and **setpos** sets the file pointer to the specified starting location of a line within the file. The beginning of each line of a text file is denoted by a block number and a byte offset into that block. Each block contains 512 bytes. The first line of a file starts with block 1, offset 0. If you try to access a nonexistent position or a position in the middle of a line, an I/O error will result.

The block number and byte offset must be values returned by **getpos**. You cannot compute the values yourself. When the file is being read, use **getpos** to determine the starting position of the next line and save that block and offset combination for later use by **setpos**.

Bear in mind that this is not "true" random access; you cannot access individual characters, only individual lines of text. Use your Pascal program to access characters individually within each line.

### Procedure 'GetPos'

Procedure **getpos** determines the starting position of the next line to be read from or written to a text file. **Getpos** requires three parameters, passed by reference, as shown below:

```
procedure GetPos(var F: text; var Block, Offset: integer);
    external;
```

where

*F* — is the file variable of type **text**.

*Block* — is the returned disk block number of the next line in file *F* to be read or written.

*Offset* — is the returned byte offset into *Block*. Together, *Block* and *Offset* point to the next line to be processed.

You should always call **getpos** to obtain the location in the file before you call **setpos**, so the block and offset values being passed to **setpos** are valid.

The example in the next subsection shows the use of **getpos**.

**Procedure 'SetPos'**

Procedure **setpos** positions the file pointer to a specified block number and byte offset into that block. **Setpos** accepts the same three parameters as **getpos**, except *Block* and *Offset* are passed by value. The **setpos** declaration is as follows:

```
procedure SetPos(var F: text; Block, Offset: integer);
    external;
```

where

| | |
|---|---|
| *F* | is the file variable of type **text**. |
| *Block* | is the block number to which the file pointer is set. |
| *Offset* | is the byte offset into *Block*. Together, *Block* and *Offset* point to the new position. |

To stress an earlier point, the block number and byte offset must be values returned by **getpos**. Do not attempt to compute the values yourself. Save the returned values for later use.

If an error is detected while **setpos** tries to position the file, the end-of-file flag **eof** is set to true. The **ioerror** and **iostatus** support library functions may help you to determine the reason that the line could not be accessed. (For details on **ioerror** and **iostatus**, see "I/O Error Trapping" in this section.) If a file is positioned to a block and offset that does not correspond to the first character of a line, the results are unpredictable.

The example below shows a way to use **getpos** and **setpos**. Program **Reverse** reads a text file and saves the position of each line in a linked list. It then prints the file in reverse line order so that the last line of the file is printed first and the first line is printed last.

```
program Reverse;

  type
    Pointer = ^position;
    position =
      record
        Next: pointer;
        Block: integer;
        Offset: integer;
      end;

  var
    F: text;
    Filename: packed array [1..80] of char;
    P, X: pointer;
    Done: boolean;

  procedure GetPos(var F: text; var Block, Offset: integer);
    external;

  procedure SetPos(var F: text; Block, Offset: integer);
    external;

  begin
    write('File name? ');
    readln(Filename);
    reset(F, Filename);
    P := nil;
    repeat                             { read the file }
      new(X);
      with X^ do GetPos(F, Block, Offset);   ———— get start of next line
      X^.Next := P;
      P := X;
      Done := eof(F);
      if not Done then readln(F);
    until Done;

    while P <> nil do                  { write the file }
      with P^ do begin
        SetPos(F, Block, Offset);   ———— set pointer to start of next line
        if not eof(F) then begin
          while not eoln(F) do begin
            write(F^);
            get(F);
            end;
          writeln;
          end;
        P := P^.Next;
        end;
end.
```

### Unsigned Integer Conversion

On the PDP-11, integer variables are stored in 16-bit words. These 16-bit words may be interpreted as signed or unsigned integers. A signed number, in two's complement notation, represents numbers in the range −32767..32767. An "unsigned" (also called "extended-range") number by definition does not have a sign bit; rather, it uses all 16 bits to represent an integer in the range 0..65535.

When their values are compared or used in mathematical expressions, unsigned integers differ greatly from signed integers. As an example, consider a word in which all 16 bits are set to one. This word has a value of -1 when interpreted as a signed integer, or a value of 65535 when interpreted as an unsigned integer. When this word is compared with some other value, the PDP-11 uses different combinations of instructions for signed and unsigned comparisons. If this number is multiplied by two, the result is a value of -2 for signed or 131070 for unsigned. The latter is an overflow condition because the result does not fit within 16 bits.

The Pascal–2 compiler and support library also differ in their treatment of signed and unsigned integers. When you define a variable to be of type **integer** in your Pascal program, the compiler treats that value as a signed integer, unless you specify an unsigned integer using a subrange notation such as:

```
type
  unsigned = 0..65535;

var
  X: unsigned;
```

According to your data declarations, the compiler will generate the correct code to compare, multiply, or divide unsigned numbers. The compiler can then deal with unsigned integers.

However, if you attempt to write out the value of an unsigned integer, you will find that the number is always treated as a signed integer. This occurs because the Pascal support library uses a single routine to print integers. This routine interprets all integers as signed values. If you want to write out the value of an unsigned integer, use the following procedure in your program instead of the **write** statement. This procedure, **Uwrite**, takes an unsigned integer and a field width as arguments. The number is printed as a value in the range 0..65535, right justified in the specified field.

```
procedure Uwrite(X: unsigned;
                 Width: integer);

  { This procedure writes an unsigned integer to output. }

  begin  { Uwrite }
    if (X > 32767) and (Width >= 0) then
      begin
      if Width > 0 then
        Width := Width - 1;
      write(X div 10: Width);
      X := X mod 10;
      Width := 1;
      end;
    write(X: Width);
  end;  { Uwrite }
```

The PDP-11 floating-point hardware uses signed conversion when it converts an integer value to a **real** value. If you wish to convert an unsigned integer to **real**, use the following function. This

function, **Ufloat**, takes an unsigned integer as its argument and returns a **real** value in the range of 0.0 to 65535.0.

```
function Ufloat(X: unsigned): real;

  { This function converts an unsigned number to a real number. }

  var
    R: real;

  begin  { Ufloat }
    R := X;
    if R < 0.0
      then R := R + 65536.0;
    Ufloat := R;
  end;  { Ufloat }
```

The **trunc** and **round** functions convert **real** numbers to integers. Since the floating-point hardware assumes a signed conversion, the following function should be used when an unsigned integer result is desired. The function **Utrunc** takes a real number in the range 0.0 to 65535.0 and converts it to an unsigned integer.

```
function Utrunc(R: real): unsigned;

  { This function converts a real number to an unsigned integer. }

  begin  { Utrunc }
    if (R > 65535.0) or (R < 0.0)
      then writeln('Unsigned number out of range');
    if R > 32767.0
      then R := R - 65536.0;
    Utrunc := trunc(R);
  end;  { Utrunc }
```

The unsigned **round** function is very similar to the above unsigned **trunc** function.

## Compiler Optimizations

The Pascal-2 compiler implements these optimizations:

### Variable Assignments to Registers

The compiler permanently assigns up to three floating-point accumulators and two general registers to commonly used local variables in each block. The compiler assigns the registers to the variables that are the most often used. No register is assigned for variables passed to a procedure as a **var** parameter or referenced directly by a procedure local to the declaring procedure. In addition, this optimization is disabled for the main program if any external procedures are referenced, since the compiler cannot determine what variables may be used by such routines.

### Assignment of Constants and Addresses to Registers

The compiler attempts to fill all registers with useful operands during compilation of a procedure, since operations on registers are faster and take less space than the corresponding operation performed in memory. Once a procedure is compiled, unused registers are filled with constant operands and addresses if such assignment saves space. This low-level optimization often results in a saving in execution time as well.

### Constant Folding

The compiler directly evaluates (folds) simple arithmetic involving constant operands of the types **integer**, **char**, **real**, and **boolean**. The generated code contains the result rather than the expression. (The RT-11 SJ compiler does not fold **real** constants; the XM compiler does.) Set expressions and relational expressions are not folded.

### Dead Code Elimination

**If** statements and **case** statements are optimized if the selection expression is constant. In this case only one path of execution is possible, and the compiler discards others. Knowledge of this optimization can lead to the writing of conditional code much like that available in some preprocessors. For example:

```
if Debugging then writeln(SomeUserValue);
```

No code for this statement is generated if the identifier **Debugging** is defined as a constant with the value **false**.

The **debug** compilation switch disables this optimization.

### Boolean Expression Optimization

When appropriate, Pascal-2 uses the minimum number of operations necessary to compute the final value of operands in Boolean expressions, thereby reducing the cost of evaluating individual Boolean expressions. (This method is known as a "short-circuit" evaluation.) The programmer must be careful not to assume that all operands of Boolean operators are evaluated or that some may not be evaluated. (This optimization takes advantage of a provision in the Pascal standard that allows an implementation to evaluate only the necessary operands of a Boolean expression.) Also, the order in which the operands are evaluated is unpredictable.

### Expression Targeting

The compiler can determine from context where a particular expression result should be computed. For instance, procedure parameters can often be computed directly on the run-time stack, and at times, expressions on the right side of the assignment operator can be computed directly into the variable on the left side.

### Common Subexpression Elimination

Multiple occurrences of the same expression are detected and simplified. Such optimization of redundant expressions is needed even though a programmer can often avoid writing such code by introducing auxiliary variables. For instance, this example:

```
writeln(I + 1, I + 1);
```

may be simplified to:

```
J := I + 1; writeln(J, J);
```

The simplification avoids the redundant computation. However, redundancy of the sort shown in the first example often leads to a more readable program. Also, certain classes of redundant expressions cannot be eliminated in the source program. For instance, array index calculations involve several underlying operations that are not reflected in the source code and therefore cannot be simplified by the programmer. Pascal–2 eliminates a wide class of common subexpressions, across statement boundaries as well as within simple expressions.

The **debug** compilation switch disables this optimization.

### Common Branch Tail Elimination

In some cases the compiler generates several branches to the same location in the object program. At times the compiler can replace redundant instructions preceding one such branch instruction with a branch to a point in the generated code that executes the same instruction stream. This low-level optimization executes an extra branch instruction in order to save some space.

The **debug** compilation switch disables this optimization.

### Array Index Simplification

Index expressions of the form [*variable* + *constant*] and [*variable* − *constant*] are partially computed. The addition or subtraction of the constant operand is folded into the value computed for the base of the array. This optimization is enabled **only** if array bounds checking is disabled and the array is unpacked.

# Appendix A: Compilation Error Messages

**'(' expected**
Check parameter list syntax.

**')' expected**
Check parameter list syntax.

**',' expected**
Check parameter list syntax.

**'..' expected**
Check array specification.

**':' expected**
Check type or **var** specification.

**':=' expected**
Check for undefined procedure or missing colon.

**';' expected after procedure body**
Use semicolons to separate procedure declarations.

**'=' expected**
Check constant or type syntax.

**'[' expected**
Check array index specification.

**']' expected**
Check array index or set specification.

**']' or ',' must follow index expression**
Check array index specification.

**A TYPE identifier is not allowed here**
The compiler encountered a bad structured constant or misplaced identifier.

**Actual parameter cannot be used with this conformant array parameter**
When conformant array parameters are used, the actual parameter must be an array and its type must be compatible with the formal parameter type.

**Actual parameter type doesn't match formal parameter type**
Parameters being passed to procedures must have the same type names as the declared (formal) parameters.

**All parameters in a single parameter section must have the same type**
Parameters grouped under one conformant array schema must be of the same type.

**Ambiguous switch**
The specified command-line switch name does not contain enough characters to distinguish it from switches with similar names.

**Array exceeds addressable memory**

**Array subscript out of range**

**Assignment of file variables not allowed**
An attempt has been made to assign an expression to a file variable or one file variable to another.

**Assignment operands are of differing or incompatible types**
Type mismatch — compare left and right sides of assignment statement for compatibility. Note that pointer types must point to identical data structures.

**Assignment to constants not allowed**

**Assignment value out of range**

**Bad adjust offset value in procedure <*name*>/main program**
A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

**Bad CASE label**
**Case** labels and case selectors must be of the same type. A colon, erroneously placed after the keyword **otherwise**, can also cause this error.

**Bad IN operands**
The left operand must be of a scalar type; the right operand must be of a compatible set type.

**Bad ORIGIN value**
**Origin** values are restricted to the I/O page (locations 0 to 1000 octal) or the system area (28K to 32K).

**Bad constant**

**Bad file name syntax**
Check command-line syntax.

**Bad parameter element**
The indicated parameter element was not followed by a ',' or a ')'.

**Bad type syntax**

**Badly formed expression**
Check parentheses and operator placement.

**BEGIN expected**
The statement part of a block must start with **begin**. Modules with no main program require the **nomain** compilation switch.

**Binary operator expected**
Two operands must be separated by an operator. Also check for mismatched quotesor an illegal digit in the cross-hatch '#' integer form.

**Block declarations are incorrectly ordered**
The relaxed ordering of declarations is an extension to standard Pascal and may be used only for global declarations.

**Block ended incorrectly**

**Block must begin with LABEL, CONST, TYPE, VAR, PROCEDURE, FUNCTION, or BEGIN**

**Boolean value expected**

**Can't assign a real value to an integer variable (use TRUNC or ROUND)**

**Can't pack unstructured or named type**

**CASE label defined twice**

**CASE label does not match selection expression type**

**CASE label must be non-real scalar type**

**CASE label type does not match tag field type**

**CASE selection expression must be a non-real scalar type**

**Code too complex in procedure <*name*>/main program**
The named body of code is too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Compiler writer error -- please contact Oregon Software at (503) 226-7760**
This indicates an internal compiler error — please save all listings and terminal output.

**Conflicting switches specified**
Certain switch combinations cannot be specified together on the compiler command line. The **debug** switch conflicts with both the **profile** and **errors** switches; the **profile** switch conflicts with the **errors** switch; and the **object** switch conflicts with the **macro** switch.

**Declaration terminated incorrectly**

**Declared labels must be defined in procedure body**

**DO expected**
Check **for**, **while** or **with** statement syntax.

**END expected**

**Exponent must lie in range -38..38**

**Expression type is incompatible with FOR index type**
For statement index types must be non-real scalars.

**External procedures/functions must be defined at outermost level**
External procedures may not be defined within other procedures.

**Extra END following block -- Check BEGIN ... END pairing**

**Extra procedures found after main program body**
This error occurs when more than one main program body (starting with a **program** statement) appears in the source file.

**Extra statements found after end of program**
This error occurs when more than one main program body appears in the source file, or when the **nomain** compilation switch is used with a source file that contains a main program body.

**Field variable expected for NEW**
Additional parameters to **new** must be tag-field constant values that identify the particular variant record being allocated.

**File cannot contain a file component**
An element of a file cannot itself contain a file.

**File names in RESET/REWRITE are non-standard**
This error is generated only when the **standard** compilation switch is enabled.

**File variable expected**
The first parameter to **reset**, **rewrite**, **get**, **put**, and **seek** must be a file variable.

**File variable or pointer variable expected**
The indicated caret (^) has been incorrectly placed after a variable that was neither a pointer nor a file.

**Files must be passed as VAR parameters**

**FOR-loop control variable can only be a simple non-real scalar variable**

**FOR-loop control variable must be declared at this level**
A **for** statement control variable must be declared local to the block containing the **for** statement.

**Format expression must be of type INTEGER**
Field-width specifications in **write** or **writeln** statements must be integers.

**Forward procedure/function body is never defined**

**Forward type reference is never resolved**
The type referenced in a pointer type declaration is not defined by later declarations.

**Function cannot be applied to an operand of this type**
A standard function has been passed a parameter of the wrong type (for example, **trunc/round** can only be applied to **real** types).

**Function identifier is never assigned a value**

**Function name expected**

`Function result must be of scalar or pointer type`
Functions may not return structured types such as records and arrays. Use **var** parameters to do this.

`Function result type cannot be duplicated in forward-declared function body`
The parameter list and result type are already specified by the forward declaration and may not be repeated. Instead, simply give the function name.

`Identifier cannot be redefined or defined after use at this level`
The specified identifier is already defined in the current block and cannot be assigned a new meaning in the indicated block.

`Identifier expected`
The indicated argument should be a variable, not a constant or expression.

`Illegal character`

`Illegal comparison of record, array, file, or pointer values`
Pointer types may be compared only for equality; record, array, and file types may not be compared in any case except strings.

`Illegal function assignment`

`Illegal subrange`
The lower bound of a subrange is required to be less than or equal to the upper bound.

`Index expression type does not match array declaration`

`Index must be non-real scalar type`

`Index variable missing in this FOR statement`

`Integer label expected`

`Integer overflow or division by zero`

`Integers must lie in range -32767..32767`

`Internal temp error in procedure` <*name*>`/main program`
A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

`Label cannot be redefined at this level`
Labels may be redefined within nested procedures, but not at the same level.

`Label defined twice`

`Label is target of illegal GOTO`
Branching into **if-then-else** or **case** statements is illegal.

`Label must be declared in LABEL declaration`

`Label must be unsigned integer constant`

`Line too long`
The maximum input line length is 160 characters.

`Listing requested but no file provided`
Check command-line syntax.

`More than two output file specifications`
Check command-line syntax.

`Must assign value before using variable`
The standard states that variables must be initialized before they are used.

`Must use VAR parameters with NONPASCAL directive`
The calling sequence for `nonpascal` procedures and functions accepts only call-by-reference parameters.

`Need at least 1 digit after '.' or 'E'`
Check for proper real numeric format.

`Need at least one value to WRITE`

`Need at least one variable to READ`

`No file in field`
Check command-line syntax.

`No input file provided`
Check command-line syntax.

`"NO" not allowed on this switch`

`No strict inclusion of sets allowed`
The operators '<' and '>' may not be applied to set operands. Instead, use '<=' or '>='.

`Non-decimal integers are not standard Pascal`
This message is issued only when the `standard` compilation switch is specified and the cross-hatch '#' integer form is used.

`Non-standard comment form, please use "{" or "(*"`
The comment form '/*', '*/' is not accepted by Pascal–2. The PASMAT utility automatically converts non-standard comments to the standard form.

`Nonsense discovered after program end`
Extraneous characters are present in the input file after the proper end of the program.

`Octal constant contains an illegal digit`
Octal constants cannot contain an 8 or a 9.

**Octal constants are not standard Pascal.**
This message is issued only when the **standard** compilation switch is specified and the conventional octal form containing 'B' is used.

**OF expected**
Check file or set declaration syntax, or **case** statement syntax.

**Only 15 levels of nesting allowed**
The compiler's limit for procedure and function nesting has been exceeded.

**Only functions can be called from expressions**
Procedures do not return a value and may not be called from within expressions.

**Operand expected**

**Operands are of differing or incompatible type**

**Operator cannot be applied to these operand types**
Check the indicated expression for proper form and operand type compatibility. For example, characters may not be multiplied together.

**OTHERWISE/ELSE clause in CASE not allowed**
**Otherwise** is an extension to standard Pascal. This message is issued when the **standard** compilation switch is specified.

**Out of memory in procedure <*name*>/main program**
The named body of code is too large or too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Output requested but no file provided**
Check command-line syntax.

**Packed array [1..n] of characters expected**
The file name arguments in **reset** and **rewrite** must be strings.

**Packed conformant array parameters cannot be nested**
Only one index type specification is allowed for packed conformant array parameters.

**Parameter list cannot be duplicated in forward-declared procedure/function body**
The indicated statement should simply give the procedure name and no parameters.

**Pointer variable expected**

**Procedure name expected**

**Procedures cannot be followed by type definition**
The relaxation of declaration ordering applies only to global declarations, and to declarations in inner blocks which precede procedure and function definitions. The indicated declaration section is improperly placed.

**PROGRAM heading expected**
This error occurs only if the **standard** compilation switch is set.

**Radix of non-decimal constant must lie in range 2..16**

**READLN and WRITELN must be applied to text file**

**Reassignment of FOR-loop control variable not allowed**
The control variable of a **for** statement may not be modified inside the body of the **for** statement.

**Record identifier expected**
A **with** statement must specify a record variable.

**Required parameter missing**
The indicated command-line switch requires a parameter as part of the switch. Consult the section in this manual on compilation switches for the required parameter.

**Same switch used twice**
Check command line for duplicate switches.

**Set is constructed of incompatible types**

**Set types must have a base in the range 0..255**

**Sets must be non-real scalar type**
The indicated set definition contains an illegal component type.

**Statement ended incorrectly**

**String constants may not include line separator**
A closing single quote (') is missing.

**String of length zero**
Strings must contain at least one character.

**Tag does not appear in variant record label list**
The tag field referred to does not exist.

**Tag identifier already used in this record**
Field identifiers within a record are required to be unique and may not be redefined within that record.

**The divisor of a MOD must be greater than zero**

**THEN expected**
Check **if** statement form.

**This function was declared as a forward procedure**
Conflict between declaration and use of function identifier. Check previous declaration.

**This parameter cannot be followed by a format expression**
A format expression may appear only in calls to **write** and **writeln**.

**This procedure was declared as a forward function**
Conflict between declaration and use of procedure identifier. Check previous declaration.

**This procedure/function name has been previously declared forward**
A procedure cannot be both **forward** and **external**, or both **forward** and **nonpascal**.

**TO or DOWNTO expected**
Check for statement syntax.

**Too few actual parameters**
The indicated parameter list does not agree with the procedure or function parameter definition.

**Too many actual parameters**
The indicated parameter list does not agree with the procedure or function parameter definition.

**Too many errors!**
The compiler error table holds 50 error messages. Error processing is terminated. Correct earlier errors and recompile for further checking.

**Too many external references in procedure <*name*>/main program**
Programs are limited to 256 external procedure references. See Appendix C of this guide for more information.

**Too many forward references (only 50 allowed)**

**Too many identifiers (only 1597 allowed)**

**Too many keys in procedure <*name*>/main program**
The named body of code is too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Too many labels in procedure <*name*>/main program**
The limit of 280 **case** labels has been exceeded in named body of code. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Too many nested INCLUDE directives (only 8 allowed)**

**Too many nodes in procedure <*name*>/main program**
The named body of code is too large to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Too many Pascal labels in procedure <*name*>/main program**
More than 32 statement labels have been declared in named body of code. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Too many procedures (only 300 allowed)**

**Too many strings or identifiers**
Restructure the program to reduce its complexity.

**Too much object code in procedure <*name*>/main program**
The named body of code is too large or too complex to be compiled. Restructure the program to reduce its complexity. See Appendix C of this guide for more information.

**Two file names in one field**
Check the command line for missing '=' or ','.

**`Travrs build error in main program`**
A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

**`Travrs walk error in main program`**
A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

**`Type name expected`**
The first parameter passed to the **`loophole`** function must be a type name.

**`Unable to open compiler scratch files`**
The disk in use does not have enough free space to store the compiler's scratch files. Try assigning `WK`: to a different disk.

**`Unary '+' or '-' cannot be applied to set operands`**

**`Undefined identifier`**

**`Undeleted temps in procedure`** *<name>* **`/main program`**
A compiler consistency-check error; please file a Trouble Report immediately. See Appendix C of this guide for more information.

**`Unexpected ')' -- Check for matching parenthesis`**

**`Unexpected ELSE clause -- Check preceding IF for extra ';'`**

**`Unknown directive`**
The legal directives are **`%include`** and **`%page`**.

**`Unknown switch`**
Check command line for error.

**`UNTIL expected`**
Check **`repeat`** statement for proper form.

**`Use '.' after main program body`**
The indicated terminator is missing from **`end`** statement.

**`Use ';' to separate declarations`**
In addition to flagging the usual missing-semicolon errors, this message is issued when an illegal digit for the specified radix is found in the cross-hatch '#' format for constants.

**`Use ';' to separate statements`**

**`Value for qualifier out of range`**
A value supplied with the indicated command-line switch is not within the allowable range for that switch.

**VAR parameters cannot be passed an expression or packed field**

A **var** parameter must be the name of a variable or a component of a data structure. If the parameter is a component of a data structure (**record** or **array**), the structure may not be packed.

**Variable name expected**

**Variable of type ARRAY expected**

**Variable of type RECORD expected**

**Variables of this type are not allowed in READ**

Scalar variables may not be used in either **read** or **write** to a text file. Only predefined types (except **boolean**) and strings may be read from a text file.

**Variables of this type are not allowed in WRITE**

Scalar variables may not be used in either **read** or **write** to a text file. Only predefined types (except **boolean**) and strings may be read from a text file.

**Variant label is undefined**

## Appendix B: Run-Time Error Messages

2 `Array subscript out of bounds`
An array index is outside of the limits established for the array in the **type** declaration that defines the array.

44 `Attempt to access block > 65535`
RT-11 files cannot contain more than 65535 blocks.

17 `Attempt to read past end of file`
An input operation was attempted on a file when `eof` is true. This is usually due to a logic error in the program and can often be solved installing checks for `eof`. This error can be trapped with the `noioerror` procedure.

15 `Attempt to write past end of file`
Files cannot be dynamically expanded. To increase the number of blocks allocated to the file, specify a larger value for the fourth parameter on the `rewrite` statement that opens the file.

33 `Attempted reference through NIL pointer`
A pointer variable was improperly used while its value was undefined or `nil`. This error could be the result of a pointer being disposed of before it is used, or of a value never being assigned to it. This could also occur if the pointer was created with `loophole` or `ref`. The `$nopointercheck` switch suppresses this error message.

41 `Can't delete file`
The specified file cannot be deleted. This error can be trapped with the `noioerror` procedure.

11 `Can't open file`
The file could not be opened for the reason identifed by the I/O error code. For input files, this error usually occurs if the file does not exist. You can trap this error by specifying and checking the fourth parameter on the `reset` or `rewrite` statement used to open the file.

42 `Can't rename file`
The file could not be renamed for the reason given by the I/O error code. This error can be trapped with the `noioerror` procedure.

37 `CASE selector matches no label`
A **case** selector expression has no matching **case** label. The **otherwise** clause can be used to detect this error. The `$norangecheck` switch disables the detection of this error.

30 `Compiler/library mismatch`
The compiler version used to compile the main program does not match the support library used when the program was linked. This error could occur if a new version of Pascal–2 is installed on your system, and you attempt to link a program with a module compiled with an older version of the compiler. The solution here is to recompile all of your modules. This error could also happen if the compiler or library were updated independently. You might have to rebuild the compiler for your system.

35 `DISPOSE() of a NIL pointer`
The pointer value does not point into the heap memory pool. This error could occur if the pointer is not initialized (via `new`) or if the pointer was created with `loophole` or `ref`.

5 **Division by zero**
Division by zero is not defined.

22 **Double deallocation of dynamic memory**
The pointer variable points to an area of memory already available for reuse. Possibly the pointer was not initialized, or it was created with **loophole** or **ref**. Also, the heap may have been corrupted. This can happen if you make assignments using uninitialized pointers or if an **external** procedure is called and the number of parameters passed to the procedure differs from the procedure definition.

18 **Error reading file**
An I/O error was detected while your program was reading an input file. The I/O error code describes the exact cause of the error. This error is most often reported during a **read** or a **get** operation. This error can be trapped with the **noioerror** procedure.

19 **Error writing file**
An I/O error was detected while your program was writing an output file. The I/O error code describes the exact cause of the error. This error is most often reported during a **write** or put operation. This error can be trapped with the **noioerror** procedure.

8 **EXP() overflow**
The parameter passed to the **exp** routine would cause an overflow condition during the calculation of the **exp**. The maximum value permitted is approximately 88.

-- **Fatal initialization error**
This error indicates that the Pascal support library could not properly initialize the program for the reason given. This error usually occurs when the program is too large.

27 **File is not a random access file. Use /SEEK**
This error is caused by an attempt to use the **seek** procedure on a file that was not opened with the **/seek** I/O control switch. The **/seek** switch must be used with the **reset** or **rewrite** statement that opened the file. This error can also occur if you attempt to open a sequential device such as a terminal or printer using **/seek**.

38 **File is not an input file**
An input operation was attempted on a file that has not been prepared for reading by **reset**. Be sure to use the **/seek** I/O control switch when you are opening random access files.

39 **File is not an output file**
An output operation was attempted on a file that has not been prepared for writing by **rewrite**.

14 **File name syntax error**
The file name is not a valid file specification. Check the file specification for invalid characters or other garbage in the file name. You can trap this error by specifying the fourth parameter on the **reset** or **rewrite** statement used to open the file.

29 **File not open**
All files other than **input** and **output** must be opened with **reset** or **rewrite** before they can be accessed.

6 **Floating point format error**
The program attempted to read a **real** number from a **text** file, where the data in the file is not a valid **real** number. This error can be trapped with the **noioerror** procedure.

3 **Floating point overflow**
The result of a floating-point operation is too large to represent as a real number. The magnitude of the largest real number is approximately $1.7E+38$.

25 **Floating point support error**
This error results from a floating-point error condition other than real overflow, integer overflow or division by zero.

32 **I/O transfer error**
A hard error (i.e., device not ready, timing error, hardware read or write error) has occurred on the channel doing the transfer. Consult the *RT-11 Software Support Manual* for further assistance.

23 **Illegal value for integer**
The program attempted to read an integer value that lies outside the range –32767..32767. This error can be trapped with the **noioerror** procedure.

9 **LOG() of zero or a negative number**
Logarithms are only defined for positive values.

-- **Multiple errors detected. Program aborted.**
This error occurs when an error is detected while another error is being processed. Rather than printing a possibly infinite list of errors, the support library prints this special error message and terminates the program. This error can be caused when the support library code has been accidentally overwritten.

21 **NEW() of zero length**
This error usually indicates an internal error in the Pascal support library. It could also be caused by an incorrect call to the **p$inew** function.

1 **Not enough memory.**
The **new** procedure was unable to allocate the requested block of memory. Dispose of noncritical memory or decrease the number of open files.

43 **Odd address or nonexistent memory trap**
An invalid memory location has been referenced. This error is the same as the PDP-11 "trap to 4" error.

40 **RENAME/DELETE of non-disk file**
The use of **rename** and **delete** is restricted to random-access devices only. Here the program attempted to perform a illegal (and meaningless) **rename** or **delete** operation on a non-disk device such as a line printer.

28 **Reserved instruction execution**
Several problems could cause this error. Check for the improper use of overlays or a mismatch between external procedure definitions and references. If this error happens on a statement involving real numbers, you may have configured your Pascal–2 system incorrectly.

26 **`SEEK() out of range`**
The program attempted to "seek" a nonexistent record. Record numbers begin with 1 and end with the last record of the file. Attempts to access record numbers less than 1 or greater than the last record (past the end of file) will cause this error.

24 **`Set element out of range`**
The program attempted to reference an element of a **`set`** that is outside the range of values permitted in the set. The valid range is 0..255.

7 **`SQRT() of a negative number`**
The square root of a negative number is undefined.

37 **`Stack overflow`**
This error could be caused by an overly large program, excessive use of dynamic memory, or deeply nested recursion. If appropriate, try overlaying the program, or close unused files and dispose of unused memory.

16 **`Too many files open`**
No more channels are available to the program. Sixteen channels are normally available (0 through 15) unless the program is overlaid, in which case the operating system uses channel 15 for overlays. Close any unused files.

20 **`TRUNC/ROUND overflow`**
The result of a **`trunc`** or **`round`** operation is too large to be represented. Only real numbers in the range −32767.0 to 32768.0 may be converted to integers with the **`trunc`** or **`round`** functions.

-- **`Unknown Pascal run-time error #`*`num`***
This message indicates that the detected error has no corresponding error message text. This indicates an internal error in the support library. Contact Oregon Software or file a Trouble Report.

10 **`Unrecognized file switch`**
An I/O control switch specified on a **`reset`** or **`rewrite`** statement is unknown to the file system. Check the spelling of your file switches. You can trap this error by specifying the fourth parameter on the **`reset`** or **`rewrite`** statement that opened the file.

34 **`Variable subrange exceeded`**
The program attempted to assign a value to a variable that is outside the bounds of the subrange type. This error is often caused by uninitialized variables or the improper use of subrange definitions. The **$norangecheck** switch disables the detection of this error.

# Appendix C: Compiler Errors

## Overflow Errors

Very complex or very large programs may exceed the capacity of the Pascal–2 compiler. Overflow of this sort is reported directly to the terminal rather than to the listing or error file. The compiler reports the type of overflow along with the name of the procedure causing the problem. Overflow errors may also occur in the main program. The following list of error messages assumes that a procedure named **MuchTooComplicated** has caused an overflow:

```
Too many keys in procedure MuchTooComplicated
Out of memory in procedure MuchTooComplicated
Too many labels in procedure MuchTooComplicated
Too many nodes in procedure MuchTooComplicated
Code too complex in procedure MuchTooComplicated
Too much object code in procedure MuchTooComplicated
Too many Pascal labels in procedure MuchTooComplicated
Too many external references in procedure MuchTooComplicated
```

An overflow condition in the main program will be reported as:

```
Too many keys in main program
```

If compilation of a program causes one of the above error conditions, simplify the offending procedure or main program section. Two suggested ways to do this are to split the routine into several sub-procedures and/or reduce the number of type definitions.

## Consistency Checks

In addition to the above error messages, consistency checks within the compiler can (in theory) trigger one of these errors:

```
Undeleted temps in main program
Internal temp error in main program
Travrs build error in main program
Travrs walk error in main program
Bad adjust offset value in main program
nnn consistency checks detected
```

You should seldom, if ever, see consistency-check errors; they are documented here for the sake of completeness. If you do see such an error, please send us a Trouble Report immediately. Along with the Trouble Report, send us the smallest possible source program that reproduces the error. Programs longer than one page should be sent on floppy disk or magnetic tape. (You also may call Oregon Software at 503-226-7760, but we will undoubtedly need to have the problem in writing.)

## Appendix D: Default File Extensions

The default file extensions listed here apply to files generated by and/or referenced by Pascal-2 and its utilities. The first column lists the type of data contained in the file. The second column lists the extension.

| File | Extension |
|---|---|
| Document | .DOC |
| Executable | .SAV |
| Listing | .LST |
| MACRO-11 Source Code | .MAC |
| Object | .OBJ |
| PROCREF Output | .PRF |
| Profiler Output | .PRO |
| PROSE Input | .PRS |
| Source | .PAS |
| Symbol Map | .SMP |
| Symbol Table | .SYM |
| Temporary | .TMP |
| XREF Output | .CRF |

## Appendix E: Entry Points in the Pascal Support Library

| Entry Point | Description |
|---|---|
| P$0 | Read character from standard file **input** |
| P$1 | Double-precision division simulation |
| P$2 | Read character from file |
| P$3 | Double-precision multiplication simulation |
| P$4 | Read integer from standard file **input** |
| P$5 | Double-precision subtraction simulation |
| P$6 | Read integer from **text** file |
| P$7 | Double-precision **real** addition simulation |
| P$8 | Read **real** number from standard file **input** |
| P$9 | Read double-precision real from standard file **input** |
| P$10 | Read **real** number from **text** file |
| P$11 | Read double-precision **real** from **text** file |
| P$12 | Read string from standard file **input** |
| P$13 | Permanently undefined (unlucky) |
| P$14 | Read string from **text** file |
| P$15 | Reserved |
| P$16 | **Readln** on standard file **input** |
| P$17 | Reserved |
| P$18 | **Readln** from **text** file |
| P$19 | Reserved |
| P$20 | Write character to standard file **output** |
| P$21 | Reserved |
| P$22 | Write character to **text** file |
| P$23 | Reserved |
| P$24 | Write integer to standard file **output** |
| P$25 | Reserved |
| P$26 | Write integer to **text** file |
| P$27 | Reserved |
| P$28 | Write **real** number to standard file **output** |
| P$29 | Write double-precision **real** to standard file **output** |
| P$30 | Write **real** number to **text** file |
| P$31 | Write double-precision **real** to **text** file |
| P$32 | Write string to standard file **output** |
| P$33 | Initialize standard files **input** and **output** |
| P$34 | Write string to **text** file |
| P$35 | Set user-handling of I/O errors for file (**noioerror**) |
| P$36 | **Writeln** to standard file **output** |
| P$37 | Status check of last file operation (**ioerror**) |
| P$38 | **Writeln** to **text** file |
| P$39 | I/O error code of last file operation (**iostatus**) |
| P$40 | Reserved |
| P$41 | "Stack overflow" error message |
| P$42 | Reserved |
| P$43 | "Subscript out of bounds" error message |
| P$44 | Reserved |

| Entry Point | Description |
|---|---|
| P$45 | "Variable subrange exceeded" error message |
| P$46 | Reserved |
| P$47 | "Reference through a nil pointer" error message |
| P$48 | Reserved |
| P$49 | "Case selector" error message |
| P$50 | Reserved |
| P$51 | "Division by zero" error message |
| P$52 | Reserved |
| P$53 | Delete file |
| P$54 | Reserved |
| P$55 | Rename file |
| P$56 | Reserved |
| P$57 | Close files in specified range |
| P$58 | Reserved |
| P$59 | Initialize Pascal |
| P$60 | Put next record |
| P$61 | Get next record |
| P$62 | Break file |
| P$63 | Program termination |
| P$64 | Rewrite file |
| P$65 | Seek record in file |
| P$66 | Reset file |
| P$67 | Debugger initialization |
| P$68 | Close file |
| P$69 | Debugger procedure entry |
| P$70 | New memory allocation |
| P$71 | Debugger procedure exit |
| P$72 | Dispose memory deallocation |
| P$73 | Debugger non-local goto |
| P$74 | Reserved |
| P$75 | Save registers |
| P$76 | Reserved |
| P$77 | Restore registers |
| P$78 | Signed integer multiply |
| P$79 | Reserved |
| P$80 | Signed integer divide |
| P$81 | Pack |
| P$82 | Signed integer mod |
| P$83 | Unpack |
| P$84 | Floating compare simulation |
| P$85 | Double-precision floating compare simulation |
| P$86 | Trunc of real number |
| P$87 | Trunc of double-precision real |
| P$88 | Float conversion to real |
| P$89 | Float conversion to double-precision real |
| P$90 | Sqrt of real number |
| P$91 | Sqrt of double-precision real |
| P$92 | Sin of real number |
| P$93 | Sin of double-precision real |
| P$94 | Cos of real number |

| Entry Point | Description |
|---|---|
| P$95 | Cos of double-precision real |
| P$96 | Atn (arctangent) of real number |
| P$97 | Atn of double-precision real |
| P$98 | Exp (exponential) of real number |
| P$99 | Exp of double-precision real |
| P$100 | Reserved |
| P$101 | Reserved |
| P$102 | Ln (natural logarithm) of real number |
| P$103 | Ln of double-precision real |
| P$104 | Reserved |
| P$105 | Reserved |
| P$106 | Time function –real |
| P$107 | Time function – double-precision real |
| P$108 | Round of real number |
| P$109 | Round of double-precision real |
| P$110 | Write boolean to standard file output |
| P$111 | Fortran interface |
| P$112 | Write boolean to text file |
| P$113 | Error reporting |
| P$114 | Reserved |
| P$115 | Reserved |
| P$116 | Unsigned integer multiplication simulation |
| P$117 | Real division simulation |
| P$118 | Unsigned integer division simulation |
| P$119 | Real multiplication simulation |
| P$120 | Unsigned integer mod |
| P$121 | Real subtraction simulation |
| P$122 | Reserved |
| P$123 | Real addition simulation |
| P$124 | Reserved |
| P$125 | Reserved |
| P$126 | Reserved |
| P$127 | Check for stack overflow |
| P$128 | Reserved |
| P$129 | Reserved |
| P$130 | Reserved |
| P$131 | Reserved |
| P$132 | Reserved |
| P$133 | Reserved |
| P$134 | Reserved |
| P$135 | Reserved |